
区块链技术指南


yeasy@github

目錄

区块链技术指南

1.1.0

区块链技术是金融科技（Fintech）领域的一项重要技术创新。

作为分布式记账（Distributed Ledger Technology，DLT）平台的核心技术，区块链被认为在金融、征信、物联网、经济贸易结算、资产管理等众多领域都拥有广泛的应用前景。区块链技术自身尚处于快速发展的初级阶段，现有区块链系统在设计和实现中利用了分布式系统、密码学、博弈论、网络协议等诸多学科的知识，为学习原理和实践应用都带来了不小的挑战。

本书希望可以探索区块链概念的来龙去脉，剥茧抽丝，剖析关键技术原理，同时讲解实践应用。在开发相关开源分布式账本平台（如超级账本），以及设计基于区块链的企业方案过程中，笔者积累了一些实践经验，也通过本书一并分享出来，希望能推动区块链技术的早日成熟和更多应用场景的出现。

阅读使用

本书适用于对区块链技术感兴趣，且具备一定信息和金融基础知识的读者；无技术背景的读者也可以从中了解到区块链的应用现状。

- 在线阅读：GitBook 或 GitHub
- pdf 版本 下载
- epub 版本 下载

进阶学习

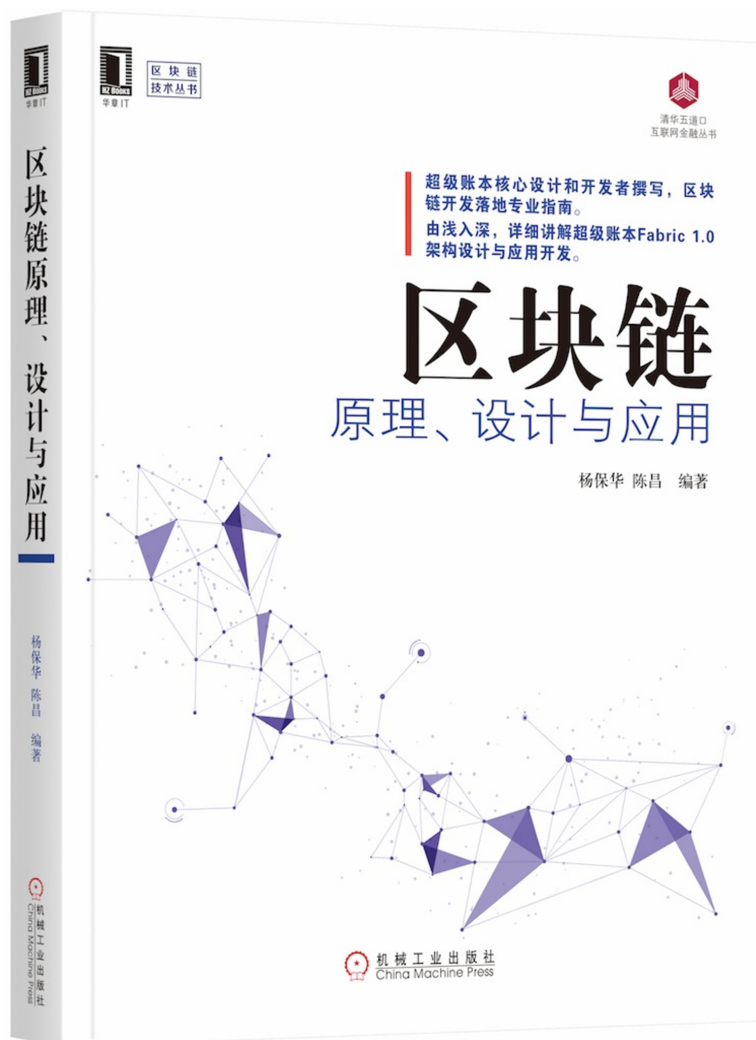


图 *1.1.1* - 区块链原理、设计与应用

《区块链原理、设计与应用》已经正式出版，详细介绍了区块链和分布式账本领域的最新技术，基于超级账本项目介绍了面向企业的分布式账本方案的设计、架构和应用，欢迎大家阅读并反馈建议。

- 京东图书
- 亚马逊图书
- China-Pub
- 当当图书

如果发现疏漏，欢迎提交到 勘误表。

参与贡献

区块链技术自身仍在快速发展中，生态环境也在蓬勃成长。欢迎 参与维护项目。

- 修订记录

- 贡献者名单

鼓励项目

欢迎鼓励项目一杯 coffee~



图 *1.1.2 - coffee*

技术交流

欢迎大家加入区块链技术讨论群：

- QQ 群 I：335626996（已满）
- QQ 群 II：523889325（已满）
- QQ 群 III：414919574（已满）
- QQ 群 IV：364824846（可加）

版本历史

- 1.1.0: 2018-04-24
    - 更新群签名；
    - 更新区块链和分布式账本演化;
    - 更新比特币、以太坊最新进展。
- 1.0.0: 2017-12-31
    - 更新 baas 设计；
    - 更新附录部分；
    - 修正部分表达。
- 0.9.0: 2017-08-24
    - 修正字词；
    - 添加 fabric 1.0 的内容；
    - 《区块链原理、设计与应用》正式出版。
- 0.8.0: 2017-03-07
    - 完善应用场景等；
    - 完善分布式系统技术；
    - 完善密码学技术；
    - 根据最新代码更新 Hyperledger 使用。
- 0.7.0: 2016-09-10
    - 完善一致性技术等；
    - 修正文字。
- 0.6.0: 2016-08-05
    - 修改文字；
    - 增加更多智能合约；
    - 增加更多业务场景。
- 0.5.0: 2016-07-10
    - 增加 Hyperledger 项目的内容；
    - 增加以太坊项目内容；
    - 增加闪电网络介绍、关键技术剖析；
    - 补充区块链即服务；
    - 增加比特币项目。
- 0.4.0: 2016-06-02
    - 添加应用场景分析。

- 0.3.0: 2016-05-12
    - 添加数字货币问题分析。
- 0.2.0: 2016-04-07
    - 添加 Hyperledger 项目简介。
- 0.1.0: 2016-01-17
    - 添加区块链简介。

参与贡献

贡献者 名单。

区块链技术自身仍在快速发展中，生态环境也在蓬勃成长。

本书源码开源托管在 Github 上，欢迎参与维护：github.com/yeasy/blockchain_guide。

首先，在 GitHub 上 `fork` 到自己的仓库，如 `docker_user/blockchain_guide`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/blockchain_guide.git
$ cd blockchain_guide
$ git config user.name "yourname"
$ git config user.email "your email"
```

更新内容后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

最后，在 GitHub 网站上提交 pull request 即可。

另外，建议定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/blockchain_guide
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

区块链的诞生

新事物往往不是凭空而生，其发展演化也不会一蹴而就。

认识一个从未见过的新事物，最重要的是要弄清楚它的来龙去脉。知其出身，方能知其所以然。

区块链（Blockchain）结构首次为人关注，来源于 2009 年初上线的比特币（Bitcoin）开源项目。然而，正如很多创新成果并非凭空出现，区块链实际上是记账科技发展到分布式场景下的天然结果。

本章将从数字货币的历史讲起，简要介绍区块链思想诞生的摇篮——比特币项目的诞生和发展过程，并初步剖析区块链技术带来的潜在的商业价值。通过阅读本章内容，读者可以了解到区块链技术出现的背景、原因，以及在诸多商业应用场景中的潜在价值。

TODO

记账科技的千年演化

如果说金融科技（Financial Technology，Fintech）是保障社会文明的重要支柱，那么记账科技（Ledger Technology，或账本科技）则是最核心的支柱基石。

无论是大到国际经济贸易，还是小到个人日常消费，都离不开记账这一看似普通、却不简单的操作。无论是资金的流转，还是资产的交易，实际上都是通过银行、交易机构中的账本记录来实现的。

可以毫不夸张地说，整个人类文明的发展历程，都伴随着账本科技的不断演化。

然而，关于账本科技的演化过程，目前还未见到相关研究，这也导致了人们对其认知的局限。特别近年来，以区块链为基础的分布式账本技术飞速崛起并得到快速应用，尽管如此，却很少有人能说清楚区块链到底因何而来，它解决了账本科技中的哪些问题，以及为何它能在金融领域产生如此巨大的影响。

笔者尝试按照科技创新的特点，将账本科技从古至今的演化过程大致分为四个阶段：简单账本、复式账本、数字化账本、分布式账本。各个阶段的时期和特点如下表所示。

阶段	时期	主要特点
阶段一：简单账本	约公元前 3500 年 ~ 15 世纪	使用原始的单式记账法（Single Entry Bookkeeping）
阶段二：复式账本	15 世纪 ~ 20 世纪中期	现代复式记账法（Double Entry Bookkeeping）出现和应用
阶段三：数字化账本	20 世纪中期 ~ 21 世纪初	物理媒介账本演化到数字化账本
阶段四：分布式账本	2009 年至今	以区块链为代表的分布式账本相关思想和技术出现

任何创新的出现都不是孤立的。账本科技的发展也与众多商业和科技成果的出现都息息相关，特别是商业贸易、计算技术、数据处理和分析手段等，如下图所示。

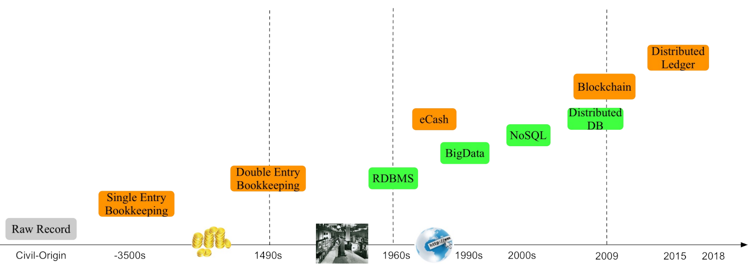


图 *1.4.1.1* - 账本科技的演化

下面笔者将具体讲述不同阶段中账本科技的发展状况。

阶段一：单式账本

在人类文明的早期，就已经有了记账的需求和活动。

最早的账本是“库辛（Kushim）泥板”，于 1929 年发掘于幼发拉底河下游右岸的伊拉克境内。据鉴定，库辛泥板属于公元前 3500 ~ 前 3000 年的乌鲁克城（Uruk，美索不达米亚西南部苏美尔人的古城），其上的文字记录内容为“37 个月收到了 29086 单位的大麦，并由库辛签核”。如下图所示。

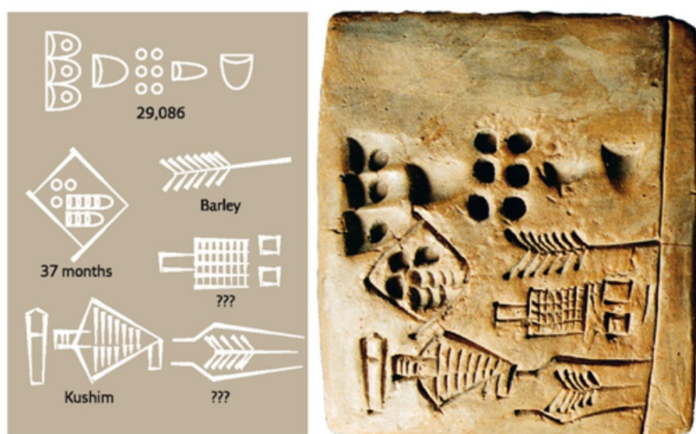


图 *1.4.1.2* - 乌鲁克城的库辛泥板

库辛泥板是目前已知的最古老的人类文字记录。另外，包括古中国、古埃及、古希腊、古罗马等古代文明，也不乏跟账本相关的考古发现。

类似通过单条记录进行账目记录的方法，被称为“单式记账法”或“简单记账法”。

此后相当长的一段时间里（至今），人们都是通过简单记账法进行记账，无论是早期记录在泥板、绳索，还是后来的纸质账本，虽然所用的物理媒介不同，但核心方法都是一致的。

简单记账法自然易用，适合小规模的简易账务，但当面对大规模、特别是涉及到多个实体的记账需求时，就暴露出了不少问题。

首先是容易出错。以库辛账本为例，如果经常发生大麦入库和出库交易，当记录很多时，很难确认账本上记录跟实际情况是否匹配；即便发现不匹配，也很难找到是哪次记录出了问题。

其次是容易被篡改。由于账本只有一个，只能保管在个人手里，假设记账者不那么诚实正直，那么，他可以轻易地通过修改已有的单条记录来从仓库中盗取大麦。并且同样的情况下，几乎没办法知道账本是否被篡改过。

随着商业活动的增多，交易规模增大和参与方的增多（特别是所有者和经营者的分离），简单记账技术已经越来越无法满足人们的需求。代表现代记账技术核心思想的“复式记账法”应运而生。

阶段二：复式账本

14 世纪的意大利，是世界贸易的门户，来自各国的商人、学者、艺术家、工匠齐聚于此，揭开了文艺复兴时期的序幕。此后长达三个世纪里，整个欧洲涌现出大量商业、文化、艺术、科技多方面的创新成果，对全世界都产生了深远的影响，例如：

- 宗教改革：马丁·路德批判了当时基督教的诸多弊端，提出宗教不应有等级制度，即宗教面前人人平等，无需任何代理人或中间人。
- 宇宙观形成：从地心说，到日心说，再到形成宇宙观，人类终于意识到自己并非所处世界的“中心”，同时任何位置也可以说是宇宙的“中心”，但这并不特别或重要。
- 复式记账法出现：将单一中心记录分拆为多个科目，提高了篡改记录的难度，并且方便追查错误根源。

这些成果分属文化、天文和金融等不同领域，但在核心思想上却如此一致和谐，不得不令人惊讶。

关于复式记账法最早的文字记载出现于 1494 年意大利著名数学家卢卡·帕切奥利（Luca Pacioli）的著作中，演化到现在包括增减记账法、收付记账法、借贷记账法三种。目前最常用的是借贷记账法，它基于会记恒等式（资产=负债+权益），确保每笔交易都按照该恒等式进行记录。复式记账法一经出现便得到了广泛的应用，成为现代会计学的重要基础。

复式记账法原理并不复杂。由于交易的本质是某种价值从来源方转移到了目标方，因此可将每笔交易分别在借方（目标方）和贷方（来源方）科目进行记录，且借贷双方的总额应该时刻保持相等。

使用复式记账法，库辛如果再收到大麦入库，会分别在“库存大麦科目”和“应收大麦科目”上都进行记录，并且两个科目的数额是一致的。如果要做审核，可以分别对不同科目进行统计，最终所有科目的结果应该保持一致。可见，使用复式记账法能很容易的追踪交易的来龙去脉，而且可以容易验证账目是否记录正确。

复式记账法解决了单个记账人对本地账本可信的问题，但是仍然无法解决多交易方不同账本的可信和互通问题。

注：借（*Debit*）：意味着债务，表示从其他方转移到本科目内；贷（*Credit*）：意味着债权，代表从该科目转移出去。

阶段三：数字化账本

如果要评价 20 世纪最伟大的十大发明，数字计算机的出现一定会入围。它在物理世界之外开创了全新的赛博空间，为社会的方方面面都带来了巨大变化。

早期的计算机，很重要的一个用途便是进行账目相关的统计处理。1951年，全世界第一台商用计算机 UNIVAC，即为美国人口普查局所使用。

使用计算机，不但可以提高大规模记账的效率，而且可以避免人工书写的错误。为了更好的管理统计数据，人们发明了专门的数据库技术。从最早的网状数据库（Network Databases）和层次数据库（Hierarchical Databases），到开创意义的关系型数据库（Relational Database），再到互联网出现后大量新需求催生的大数据、NoSQL 等技术，根源上其实都与记账息息相关。

在这一阶段，记账方法本身的思想并没有太多创新，但由于数字媒介的出现，使得账本的规模、记账处理的速度、账本的复杂度，都有了天翻地覆的提升。而这直接为后来包括电子商务、互联网金融在内的多种数字化服务奠定了核心技术基础。

注：关于 *NoSQL* 的理解曾发生数次变化：*Not SQL*！（不支持 *SQL*！）；*No*，*SQL*！（不，还得要 *SQL*！）；*Not Only SQL*（不仅仅是 *SQL*！）。

阶段四：分布式账本

复式记账法里虽然记录了每笔交易的来龙去脉、不宜错误、也十分方便统计，但仍然是中心化的模式。

中心化模式的账本结构方便使用，但在很多情况下仍然存在不少问题：账本掌握在个体手中，一旦出现数据丢失则无法找回；同时涉及到多个交易方的情况下，需要分别维护各自的账本，容易出现不一致、对账也较困难的情况。

由此可以很自然的想到，可以借助分布式系统的思想来实现分布式账本（Distributed Ledger）：由交易多方一起来共同维护同一个分布式账本；打通交易在各个阶段的来龙去脉；凭借分布式技术，进一步提高记账的效率和可靠性。但在分布式的情况下，如何避免参与方恶意篡改或破坏记录？如何决定该由谁来将交易记录写到账本中？这些问题一直没有得到很好的解答。

直到 2009 年 1 月，基于区块链结构的比特币网络悄然问世，此后数年里，在分布式的场景下稳定支持了大量的转账业务。这让人们开始认识到，区块链这一极为简洁的数据结构，居然恰好解决了分布式记账的基本需求。目前，基于区块链结构的分布式账本解决方案，也往往被简称为区块链技术。

从 2014 年开始，金融、科技领域的专家们越来越关注区块链相关的技术，尝试研发基于分布式账本技术的解决方案。

2015 年底，三十家金融、科技领域的巨头企业们（包括 IBM、Accenture、Intel、J.P.Morgan、DTCC、SWIFT、Cisco 等）联合成立了超级账本（Hyperledger）项目，由 Linux 基金会进行管理，遵循 Apache v2 许可，致力于打造一个开源、满足企业级应用的分布式账本科技生态。目前围绕企业分布式账本的核心诉求，已经发展到 9 大顶级项目，超过 200 名来自全球的企业会员，并支撑了众多的探索和落地案例。

目前，基于分布式账本技术的各种创新方案已经在包括金融、供应链、医疗等领域得到了不少落地应用。但笔者认为，类比互联网科技的发展过程，目前分布式账本技术还处于发展的初期，还存在不少尚待解决的问题，包括权限管理、隐私保护、性能优化和互操作性等。在这些方面的科技突破，可以极大拓展分布式账本技术的应用场景和形态，真正实现传递“价值”的商业互联网络。

注：超级账本项目中来自中国的企业会员已经超过 *40* 家，包括百度、腾讯、华为、万达等科技巨头。

注：互联网自诞生之日起用了十余年时间才从内部的 *ARPA* 网络演化到互联可通；又用了十余年时间才打造了包括 *TCP/IP* 等在内的基础架构和协议；在其三十岁生日时才可以开放接入，开始支持商业应用场景。

账本科技的未来

账本科技经历了数千年的演化，整体发展趋势由简单到复杂、由粗糙到精细、由中心化到分布式，这与运行在其上的业务需求的不断变化密不可分。伴随着社会的发展，大规模、高安全、易审计将始终是账本科技应用中最受人们关注的特性。

笔者相信，随着社会文明特别是商业活动的进一步的发展和成熟，分布式账本科技的需求将变得更加普遍和强烈。分布式账本科技自身也将变得更加成熟和可靠。

分布式记账与区块链

从分布式记账问题说起

分布式记账问题由来已久。自以电子计算机为主要记账手段的数字化账本诞生以来，商业活动参与者首先要寻找一个多方均能信任的第三方来记账，确保交易记录的准确。然而，随着商业活动的规模越来越大，商业过程愈加动态和复杂，很多场景下难以找到符合要求的第三方记账方（例如供应链领域动辄涉及来自多个行业的数百家参与企业）。这就需要交易各方探讨在多方分布式场景下进行协同记账的可能性。

实际上，可以很容易设计出一个简单粗暴的分布式记账结构，如下图所示方案（一）。多方均允许对账本进行任意读写，一旦发生新的交易即追加到账本上。这种情况下，如果参与多方均诚实可靠，则该方案可以正常工作；但是一旦有参与方恶意篡改已发生过的记录，则无法确保账本记录的正确性。

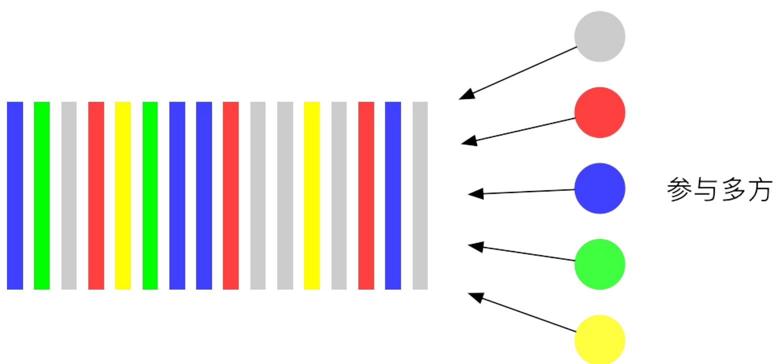


图 *1.4.2.1* - 方案（一）：简单分布式记账结构

为了防止有参与者对交易记录进行篡改，需要引入一定的验证机制。很自然地，可以借鉴信息安全领域的数字摘要（Digital Digest）技术，从而改进为方案（二）。每次当有新的交易记录被追加到账本上时，参与各方可以使用 Hash 算法对完整的交易历史计算数字摘要，获取当前交易历史的“指纹”。此后任意时刻，每个参与方都可以对交易历史重新计算数字摘要，一旦发现指纹不匹配，则说明交易记录被篡改过。同时，通过追踪指纹改变位置，可以定位到被篡改的交易记录。

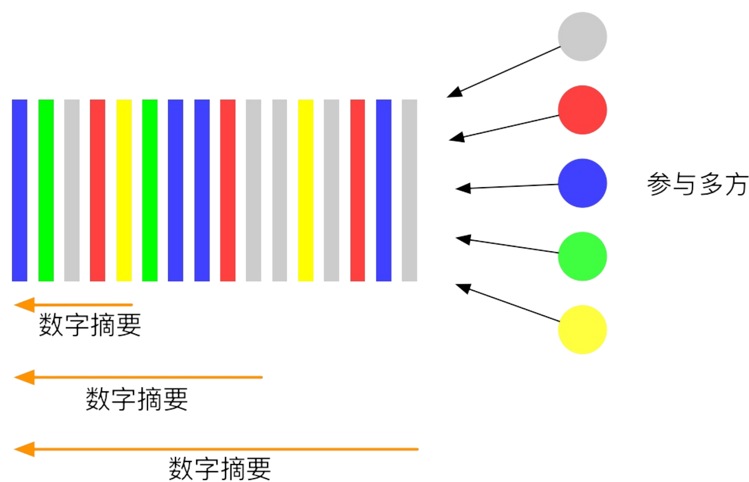


图 1.4.2.2 - 方案（二）：带有数字摘要验证的分布式记账

方案（二）可以解决账本记录防篡改的问题，然而在实际生产应用时，仍存在较大缺陷。由于每次追加新的交易记录时需要从头对所有的历史数据计算数字摘要，当已存在大量交易历史时，数字摘要计算成本将变得很高。而且，随着新交易的发生，计算耗费将越来越大，系统扩展性很差。

为了解决可扩展性的问题，需要进一步改进为方案（三）。注意到任何一次摘要实际上已经确保了从头开始到摘要位置的完整历史，当新的交易发生后，实际上需要进行增加验证的只是新的交易，即增量部分。因此，计算摘要的过程可以改进为对旧的摘要值再加上新的交易内容进行验证。这样就既解决了防篡改问题，又解决了可扩展性问题。

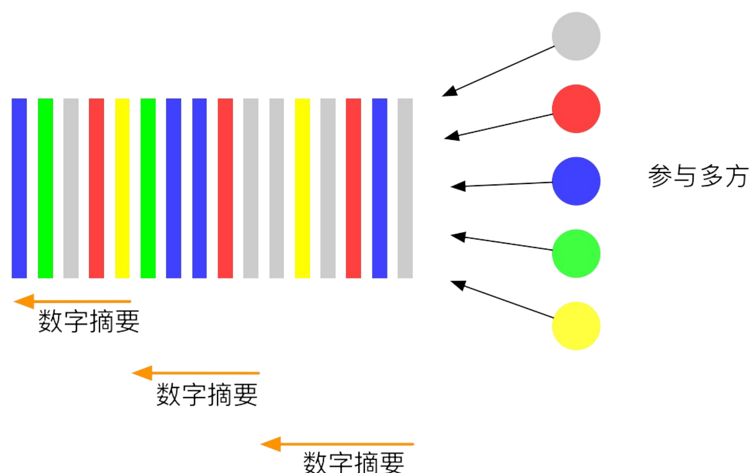


图 1.4.2.3 - 方案（三）：带有数字摘要验证的可扩展的分布式记账

实际上，读者可能已经注意到，方案（三）中的账本结构正是一个区块链结构（如下图所示）。可见，从分布式记账的基本问题出发，可以自然推导出区块链结构，这也说明了在分布式场景下的记账问题中，区块链结构是一个简洁有效的天然答案。

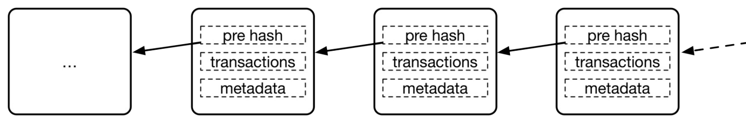


图 *1.4.2.4* - 区块链结构

区块链的三次热潮

从比特币诞生之日（2009 年 1 月）算起，区块链已在全世界掀起了三次热潮。



图 *1.4.2.5* - 区块链的三次热潮

第一波热潮出现在 2013 年左右。比特币项目上线后，很长一段时间里并未获得太多关注。直到比特币价格发生暴涨，各种加密货币项目纷纷出现，隐藏在其后的区块链结构才首次引发大家的兴趣。2014 年起，区块链这个术语开始频繁出现，但更多集中在加密货币和相关技术领域；

第二波热潮出现在 2016 年前后。以区块链为基础的分布式账本技术被证实在众多商业领域存在应用价值。尤其 2015 年 10 月《经济学人》封面文章《信任机器》中，正式指出区块链在构建分布式账本平台中的重要作用，更多实验性应用出现。下半年更是出现了“初始代币发行（Initial Coin Offering，ICO）”等新型融资募集形式。这一时期，区块链技术自身也有了突破，首个大规模公有智能合约引擎——以太坊项目正式上线；首个面向企业应用的联盟分布式账本——超级账本项目，在众多企业巨头的支持下也正式成立。

随着更多商业项目开始落地，从 2017 年开始至今，众多互联网领域的资本开始关注区块链领域，人才缺口持续加大，区块链已经俨然成为继人工智能后的又一资本热点。

分析这三次热潮可以看出，每一次热潮的出现都与金融行业对区块链技术的深化应用密切相关。这也表明金融行业对信息科技成果始终保持了较高的敏感度。

分布式记账的重要性

分布式记账问题为何重要？可以类比互联网出现后对社会带来的重大影响。

互联网是人类历史上最大的分布式互联系统。作为信息社会的基础设施，它很好地解决了传递信息的问题。然而，由于早期设计上的缺陷，互联网无法确保所传递信息的可靠性，这大大制约了人们利用互联网进行大规模协作的能力。而以区块链为基础的分布式账本科技则可能解决传递可信信息的问题。这意味着基于分布式账本科技的未来商业网络，将成为新一代的文明基础设施——大规模的协作网络。

分布式账本科技的核心价值在于为未来多方协同网络提供可信基础。区块链引发的记账科技的演进，将促使商业协作和组织形态发生变革。甚至世界经济论坛执行主席 Klaus Schwab 认为“区块链是（继蒸汽机、电气化、计算机之后的）第四次工业革命的核心成果（Blockchains are at the heart of the Fourth Industrial Revolution）”。

分布式账本的现状与未来

类比互联网，从科技发展的一般规律而言，笔者认为，分布式账本科技仍处于发展早期，而商业应用已经在加速落地中，如下表所示。

阶段	互联网	区块链	阶段
1974~1983	ARPANet 内部试验网络	比特币试验网络	2009~2014
1984~1993	TCP/IP 基础协议确立，基础架构完成	基础协议和框架探索，出现超级账本、以太坊等开源项目	2014~2019?
1990s~2000s	HTTP 应用协议出现；互联网正式进入商用领域	商业应用的加速落地，仍未出现杀手级应用	2018~？
2000s~？	桌面互联网、移动互联网、物联网	分布式协同商业网络	？

互联网在发展过程中，先后经历了试验网络、基础架构和协议、商业应用、大规模普及等四个阶段。其中第二个阶段尤为关键，TCP/IP 协议取代了已有的网络控制协议，成为基础协议栈的核心，这为后来互联网扩展到全球规模奠定了扎实的技术基础。

作为一套前所未有的大规模协同网络，分布式账本网络的发展很大可能也要经历这四个阶段的演化。当然，站在前人肩膀上，无论是演化速度还是决策效率，都会有不小的优势。

客观来看，虽然超级账本、以太坊等开源项目在基础协议和框架方面进行了诸多探索，并取得了重要成果，但在多账本互联、与已有系统的互操作性等方面还存在不足，商业应用的广度和深度仍需实践的考验。

但毫无疑问，分布式账本科技已经成为金融科技领域的重要创新成果，带来的效率提升必将为金融系统创造新的发展机遇。而互联网之后的商业协同网络，必将进一步推动整个人类文明的进步。

站在前人肩膀上的比特币

加密货币的历史

随着上世纪 50 年代计算机的出现和发展（ENIAC，1946 年），人们就尝试将信息技术应用到支付过程以提高效率。除了作为电子支付手段的银行卡，自 80 年代起，利用密码学手段构建的数字货币（加密货币）也开始成为研究的热门。

加密货币的发展前后经历了近 30 年的演化，比较典型的成果包括 e-Cash、HashCash、B-money 等。

1983 年，时任加州大学圣塔芭芭拉分校教授的 David Chaum 最早在论文《Blind Signature for Untraceable Payments》 中提出了 e-Cash，并于 1989 年创建了 DigiCash 公司。ecash 系统是首个尝试解决不可追踪（untraceable）问题的匿名数字货币，基于 David Chaum 自己发明的盲签名技术，曾被应用于部分银行的小额支付系统中。ecash 虽然不可追踪，但仍依赖中心化机构（银行）的协助，同期也由于信用卡体系的快速崛起，DigiCash 公司最终于 1998 年宣告破产。鉴于 David Chaum 在数字货币研究领域发展早期的贡献，有人认为他是“数字货币之父”。

1997 年，Adam Back 发明了 HashCash，来解决邮件系统和博客网站中“拒绝服务攻击（Deny of Service，DoS）”攻击问题。Hashcash 首次提出用工作量证明（Proof of Work，PoW）机制来获取额度，该机制后来被后续数字货币技术所采用。

1998 年，Wei Dai 提出了 B-money 的设计，这是首个不依赖中心化机构的匿名数字货币方案。B-money 引入工作量证明的思想来解决数字货币产生的问题，指出任何人都可以发行一定量的货币，只要他可以给出某个复杂计算问题（未说明是用 Hash 计算）的答案，货币的发行量将跟问题的计算代价成正比。并且，任何人（或部分参与者）都可以维护一套账本，构成一套 P2P 网络，使用者在网络内通过对带签名的交易消息的广播来实现转账的确认。B-money 是去中心化数字货币领域里程碑式的成果，为后面比特币的出现奠定了基础。从设计上看，B-money 已经很好的解决了货币发行的问题，但是未能解决“双花”问题，也未能指出如何有效、安全地维护账本，最终未能实现。

这些数字货币方案要么依赖于一个中心化的管理机构，要么更多偏重理论层面的设计。直到比特币的出现，采用了区块链结构来维护账本，并引入经济博弈机制，充分利用现代密码学成果，首次从实践意义上实现了一套去中心化的数字货币系统。也正因为比特币的影响力巨大，现在谈到数字货币往往是指类似基于密码学原理的加密货币（crypto currency）。

比特币依托的分布式网络无需任何管理机构，通过密码学原理来确保交易的正确进行；另一方面，比特币的价值和发行并未有中央机构进行调控，而是通过计算力进行背书，通过经济博弈进行自动调整。这也促使人们开始思考，在数字化的世界中，应该如何发行货币，以及如何衡量价值。

目前，除了像以比特币这样完全丢弃已有体系的分布式技术之外，仍然存在不少中心化代理模式的数字货币机制，包括类似 paypal 这样的平台。通过跟已有的支付系统合作，可以更高效率地代理完成交易。

现在还很难讲哪种模式将会成为日后的主流，未来甚至还可能出现更先进的技术。但毫无疑问，这些成果都为后来的数字货币设计提供了极具价值的参考；而站在前人肩膀上的比特币，必将在人类货币史上留下难以磨灭的印记。

比特币的诞生

2008 年 10 月 31 日（东部时间），星期五下午 2 点 10 分，化名 Satoshi Nakamoto（中本聪）的人在 metzdowd 密码学邮件列表 中提出了比特币（Bitcoin）的设计白皮书《Bitcoin: A Peer-to-Peer Electronic Cash System》，并在 2009 年公开了最初的实现代码。首个比特币是 UTC 时间 2009 年 1 月 3 日 18:15:05 生成。但比特币真正流行开来，被人们所关注则是至少两年以后了。

作为开源项目，比特币很快吸引了大量开发者的加入，目前的官方网站 bitcoin.org，提供了比特币相关的代码实现和各种工具软件

除了精妙的设计理念外，比特币最为人津津乐道地一点，是发明人“中本聪”到目前为止尚无法确认真实身份。也有人推测，“中本聪”背后可能不止一个人，而是一个团队。这些猜测都为比特币项目带来了不少传奇色彩。

比特币的意义和价值

直到今天，关于比特币的话题仍充满了不少争议。但大部分人应该都会认可，比特币是数字货币历史上，甚至整个金融历史上一次了不起的社会学实验。

比特币网络在 2009 年上线以来，在无人管理的情况下，已经在全球范围内 `7*24` 小时运行超过 8 年时间，成功处理了几百万笔交易，甚至支持过单笔 1.5 亿美金的交易。更为难得的是，比特币网络从未出现过重大的系统故障。

比特币网络目前由数千个核心节点参与构成，不需要任何中心化的支持机构参与，纯靠分布式机制支持了稳定上升的交易量。

比特币首次真正从实践意义上实现了安全可靠的去中心化数字货币机制，这也是它受到无数金融科技从业者热捧的根本原因。

作为一种概念货币，比特币主要是希望解决已有货币系统面临的几个核心问题：

- 被掌控在单一机构手中，容易被攻击。
- 自身的价值无法保证，容易出现波动。
- 无法匿名化交易，不够隐私。

在前文中也探讨过，要实现一套数字货币机制，最关键的还是要建立一套完善的交易记录系统，以及形成一套合理的货币发行机制。

这个交易记录系统要能准确、公正地记录发生过的每一笔交易，并且无法被恶意篡改。对比已有的银行系统，可以看出，现有的银行机制作为金融交易的第三方中介机构，有代价地提供了交易记录服务。如果参与交易的多方都完全相信银行的记录（数据库），就不存在信任问题。可是如果是更大范围（甚至跨多家银行）进行流通的货币呢？哪家银行的系统能提供完全可靠不中断的服务呢？唯一可能的方案是一套分布式账本。这个账本可以被所有用户自由访问，而且任何个体都无法对所记录的数据进行恶意篡改和控制。为了实现这样一个前所未有的账本系统，比特币网络巧妙地设计了区块链结构，提供了可靠、无法被恶意篡改的数字货币账本功能。

比特币网络中，货币的发行是通过比特币协议来规定的。货币总量受到控制，发行速度随时间自动进行调整。既然总量一定，那么单个比特币的价值会随着越来越多的经济实体认可比特币而水涨船高。发行速度的自动调整则避免出现通胀或者滞涨的情况。

另一方面，也要冷静地看到，作为社会学实验，比特币已经获得了巨大的成功，特别是基于区块链技术，已经出现了许多颇有价值的商业场景和创新技术。但这绝不意味着比特币自身必然能够进入到未来的商业体系中。

更有价值的区块链技术

如果说比特币是影响力巨大的社会学实验，那么从比特币核心设计中提炼出来的区块链技术，则让大家看到了塑造更高效、更安全的未来商业网络的可能。

2014 年开始，比特币背后的区块链技术开始逐渐受到大家关注，并进一步引发了分布式记账本（Distributed Ledger）技术的革新浪潮。

实际上，人们很早就意识到，记账相关的技术，对于资产（包括有形资产和无形资产）的管理（包括所有权和流通）十分关键；而去中心化或多中心化的分布式记账本技术，对于当前开放、多维化的商业模式意义重大。区块链的思想和结构，正是实现这种分布式记账本系统的一种极具可行潜力的技术。

区块链技术现在已经脱离开比特币网络自身，在包括金融、贸易、征信、物联网、共享经济等诸多领域崭露头角。现在，除非特别指出是“比特币区块链”，否则当人们提到“区块链技术”进行探讨时，往往所指已经与比特币没有什么必然联系了。

潜在的商业价值

商业行为的典型模式为：交易的多方通过协商和执行合约，完成交易过程。区块链擅长的正是如何在多方之间达成合约，并确保合约的顺利执行。

根据类别和应用场景不同，区块链所体现的特点和价值也不同。

从技术角度，一般认为，区块链具有如下特点：

- 分布式容错性：分布式网络极其鲁棒，能够容忍部分节点的异常状态；
- 不可篡改性：一致提交后的数据会一直存在，不可被销毁或修改；
- 隐私保护性：密码学保证了数据隐私，即便数据泄露，也无法解析。

随之带来的业务特性将可能包括：

- 可信任性：区块链技术可以提供天然可信的分布式账本平台，不需要额外第三方中介机构参与；
- 降低成本：跟传统技术相比，区块链技术可能带来更短的时间、更少的人力，降低维护成本；
- 增强安全：区块链技术将有利于安全、可靠的审计管理和账目清算，减少犯罪风险。

区块链并非凭空诞生的新技术，更像是技术演化到一定程度突破应用阈值后的产物，因此，其商业应用场景也跟促生其出现的环境息息相关。对于基于数字方式的交易行为，区块链技术能潜在地降低交易成本、加快交易速度，同时能提高安全性。笔者认为，能否最终带来成本的降低，将是一项技术能否被深入应用的关键。

所有跟信息、价值（包括货币、证券、专利、版权、数字商品、实际物品等）、信用等相关的交换过程，都将可能从区块链技术中得到启发或直接受益。但这个过程绝不是一蹴而就的，可能需要较长时间的探索和论证。具体的商业应用案例可以参考本书后续的应用场景章节。

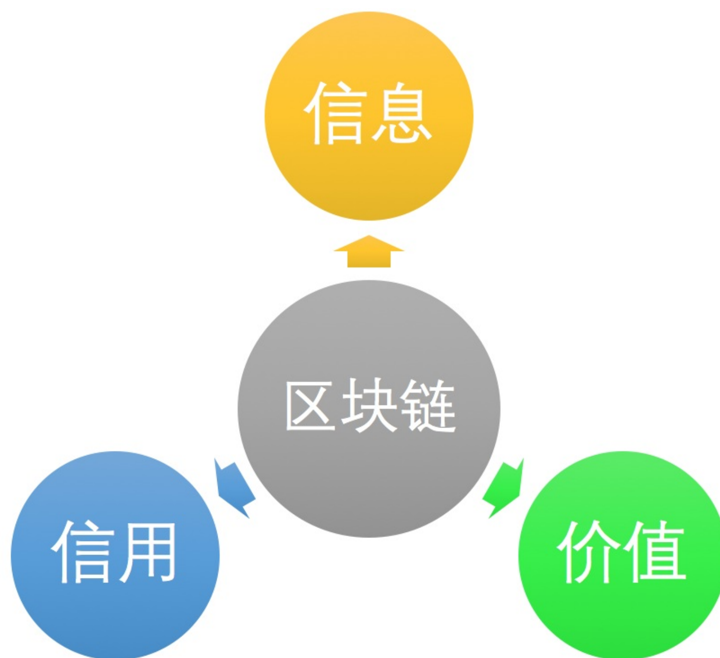


图 *1.4.4.1* - 区块链影响的交换过程

目前，区块链技术已经得到了众多金融机构和商业公司的关注，包括大量金融界和信息技术界的领军性企业和团体。典型企业组织如下所列（排名不分先后）。

- Visa 国际组织
- 美国纳斯达克证券交易所（Nasdaq）
- 高盛投资银行（Goldman Sachs）
- 花旗银行（Citi Bank）
- 美国富国银行（Wells Fargo）
- 中国人民银行
- 中国浦发银行
- 日本三菱日联金融集团
- 瑞士联合银行
- 德意志银行
- 美國的證券集中保管結算公司（DTCC）
- 全球同业银行金融电讯协会（SWIFT）
- 国际商业机器公司（IBM）
- 微软（Microsoft）
- 英特尔（Intel）
- 思科（Cisco）
- 埃森哲（Accenture）

Gartner 在 2017 年的报告《Forecast: Blockchain Business Value, Worldwide, 2017-2030》中预测："区块链带来的商业价值在 2025 年将超过 17.6 亿美金，2030 年将超过 3.1 万亿美金（the business value-add of blockchain will grow to slightly more than $176 billion by

2025, and then it will exceed $3.1 trillion by 2030）”。

本书将在后续章节中通过具体的案例来讲解区块链的多个商业应用场景。

本章小结

区块链思想诞生于数字货币长达三十多年的发展过程中，它支持了首个自带信任、防篡改的分布式记录系统——比特币网络。这也第一次让大家意识到，除了互联网这样的尽力而为（不保证可信）的基础设施外，基于区块链技术还将可能打造一个实现彼此信任的基础网络设施。

当然，从应用角度讲，比特币也只是基于区块链技术的一种金融应用。区块链技术其实还能带来更通用的计算能力和商业价值。本书后续章节将介绍更多的商业应用案例，并介绍开源界打造的更多区块链平台项目，包括以太坊和超级账本 等项目。这些开源项目加速释放了区块链技术的威力，为更多更复杂的区块链应用场景提供了技术支持。

核心技术概览

运用之妙夺造化，存乎一心胜天工。

跨境商贸合作中签订的合同，怎么确保对方能严格遵守和及时执行？

酒店宣称刚打捞上来的三文鱼，怎么追踪捕捞和运输过程中的时间和卫生？

现代数字世界里，怎么证明你是谁？怎么证明某个资产属于你？

经典囚徒困境中的两个人，怎样才能达成利益的最大化？

宇宙不同文明之间的“黑暗森林”猜疑链，有没有可能被彻底打破？

这些看似很难解决的问题，在区块链的世界里已经有了初步的答案。

本章将带领大家探索区块链的核心技术，包括其定义与原理、关键的问题等，还将探讨区块链技术的演化，并对未来发展的趋势进行展望。最后，对一些常见的认识误区进行了澄清。

定义与原理

定义

区块链技术自身仍然在飞速发展中，目前相关规范和标准还在进一步成熟中。

公认的最早关于区块链的描述性文献是中本聪所撰写的 《比特币：一种点对点的电子现金系统》，但该文献重点在于讨论比特币系统，实际上并没有明确提出区块链的术语。在其中，区块和链被描述为用于记录比特币交易账目历史的数据结构。

另外，Wikipedia 上给出的定义中，将区块链类比为一种分布式数据库技术，通过维护数据块的链式结构，可以维持持续增长的、不可篡改的数据记录。

笔者认为，讨论区块链可以从狭义和广义两个层面来看待。

狭义上，区块链是一种以区块为基本单位的链式数据结构，区块中利用数字摘要对之前的交易历史进行校验，适合分布式记账场景下防篡改和可扩展性的需求。

广义上，区块链还指代基于区块链结构实现的分布式记账技术，还包括分布式共识、隐私与安全保护、点对点通信技术、网络协议、智能合约等。

区块链技术最早的应用出现在比特币项目中。作为比特币背后的分布式记账平台，在无集中式管理的情况下，比特币网络稳定运行了八年时间，支持了海量的交易记录，并且从未出现严重的漏洞，这些都与区块链结构的巧妙是分不开的。

基本原理

区块链的基本原理理解起来并不复杂。

首先包括三个基本概念：

- 交易（Transaction）：一次对账本的操作，导致账本状态的一次改变，如添加一条转账记录；
- 区块（Block）：记录一段时间内发生的所有交易和状态结果，是对当前账本状态的一次共识；
- 链（Chain）：由区块按照发生顺序串联而成，是整个账本状态变化的日志记录。

如果把区块链作为一个状态机，则每次交易就是试图改变一次状态，而每次共识生成的区块，就是参与者对于区块中交易导致状态改变的结果进行确认。

在实现上，首先假设存在一个分布式的数据记录账本，这个账本只允许添加、不允许删除。账本底层的基本结构是一个线性的链表，这也是其名字“区块链”的来源。链表由一个个“区块”串联组成（如下图所示），后继区块记录前导区块的哈希（Hash）值。新的数据要加入，

必须放到一个新的区块中。而这个块（以及块里的交易）是否合法，可以通过计算哈希值的方式快速检验出来。任意维护节点都可以提议一个新的合法区块，然而必须经过一定的共识机制来对最终选择的区块达成一致。

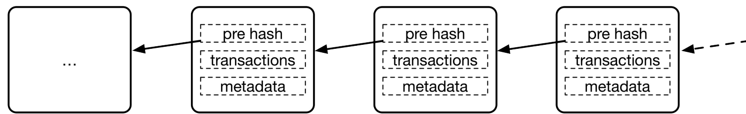


图 *1.5.1.1* - 区块链结构示例

以比特币为例理解区块链工作过程

具体以比特币网络为例，来看其中如何使用了区块链技术。

首先，比特币客户端发起一项交易，广播到比特币网络中并等待确认。网络中的节点会将一些收到的等待确认的交易记录打包在一起（此外还要包括前一个区块头部的哈希值等信息），组成一个候选区块。然后，试图找到一个 nonce 串（随机串）放到区块里，使得候选区块的哈希结果满足一定条件（比如小于某个值）。这个nonce 串的查找需要一定的时间进行计算尝试。

一旦节点算出来满足条件的 nonce 串，这个区块在格式上就被认为是“合法”了，就可以尝试在网络中将它广播出去。其它节点收到候选区块，进行验证，发现确实符合约定条件了，就承认这个区块是一个合法的新区块，并添加到自己维护的区块链上。当大部分节点都将区块添加到自己维护的区块链结构上时，该区块被网络接受，区块中所包括的交易也就得到确认。

当然，在实现上还会有很多额外的细节。这里面比较关键的步骤有两个，一个是完成对一批交易的共识（创建区块结构）；一个是新的区块添加到区块链结构上，被大家认可，确保未来无法被篡改。

比特币的这种基于算力寻找 nonce 串的共识机制被称为工作量证明（Proof of Work，PoW）。目前，要让哈希结果满足一定条件，并无已知的快速启发式算法，只能进行逐个尝试的暴力计算。尝试的次数越多（工作量越大），算出来的概率越大。

通过调节对哈希结果的限制，比特币网络控制平均约 10 分钟产生一个合法区块。算出区块的节点将得到区块中所有交易的管理费和协议固定发放的奖励费（目前是 12.5 比特币，每四年减半）。这个计算新区块的过程，即俗称的“挖矿”。

读者可能会关心，比特币网络是任何人都可以加入的，如果网络中存在恶意节点，能否进行恶意操作来对区块链中记录进行篡改，从而破坏整个比特币网络系统。比如最简单的，故意不承认收到的别人产生的合法候选区块，或者干脆拒绝来自其它节点的交易等。

实际上，因为比特币网络中存在大量（据估计数千个）的维护节点，而且大部分节点都是正常工作的，默认都只承认所看到的最长的链结构。只要网络中不存在超过一半的节点提前勾结一起采取恶意行动，则最长的链将很大概率上成为最终合法的链。而且随着时间增加，这个概率会越来越大。例如，经过 6 个区块生成后，即便有一半的节点联合起来想颠覆被确认的结果，其概率也仅为 (1/2)^6 ≈ 1.6%，即低于 1/60 的可能性。

当然，如果整个网络中大多数的节点都联合起来作恶，可以导致整个系统无法正常工作。要做到这一点，往往意味着付出很大的代价，跟通过作恶得到的收益相比，得不偿失。

注：区块链结构，与 *Git* 版本管理的有向无环图数据结构，在设计上有异曲同工之妙。

技术的演化与分类

区块链技术自比特币网络设计中被大家发掘关注，从最初服务数字货币系统，到今天在分布式账本场景下发挥着越来越大的技术潜力。

区块链的演化

比特币区块链已经支持了简单的脚本计算，但仅限于数字货币相关的处理。除了支持数字货币外，还可以将区块链上执行的处理过程进一步泛化，即提供智能合约（Smart Contract）。智能合约可以提供除了货币交易功能外更灵活的合约功能，执行更为复杂的操作。

这样扩展之后的区块链，已经超越了单纯数据记录的功能了，实际上带有点“智能计算”的意味了；更进一步地，还可以为区块链加入权限管理，高级编程语言支持等，实现更强大的、支持更多商用场景的分布式账本。

从计算特点上，可以看到现有区块链技术的三种典型演化场景：

场景	功能	智能合约	一致性	权限	类型	性能	编程语言	代表
公信的数字货币	记账功能	不带有或较弱	PoW	无	公有链	较低	简单脚本	比特币网络
公信的交易处理	智能合约	图灵完备	PoW、PoS	无	公有链	受限	特定语言	以太坊网络
带权限的分布式账本处理	商业处理	多种语言，图灵完备	包括 CFT、BFT 在内的多种机制，可插拔	支持	联盟链	可扩展	高级编程语言	超级账本

区块链与分布式记账

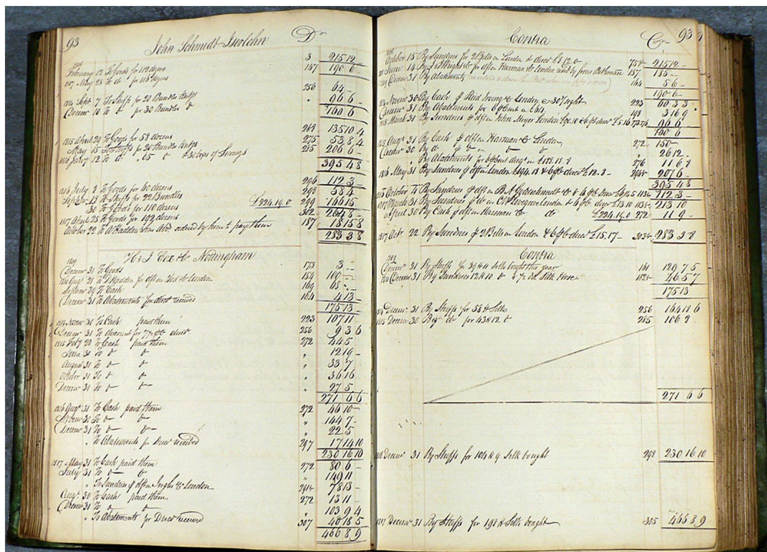


图 *1.5.2.1* - 古老的账本

记账技术历史悠久，现代复式记账系统（Double Entry Bookkeeping）是由意大利数学家卢卡·帕西奥利 1494 年在《Summa de arithmetica, geometrica, proportioni et proportionalità》一书中最早制定。复式记账法对每一笔账目同时记录来源和去向，首次将对账验证功能引入记账过程，提升了记账过程的可靠性。

从这个角度来看，区块链是首个自带对账功能的数字记账技术实现。

更广泛地，区块链属于一种去中心化的记录技术。参与到系统上的节点，可能不属于同一组织，彼此无需信任；区块链数据由所有节点共同维护，每个维护节点都能复制获得一份完整或部分记录的拷贝。

跟传统的记账技术相比，基于区块链的分布式账本应该包括如下特点：

- 维护一条不断增长的链，只可能添加记录，而发生过的记录都不可篡改；
- 去中心化，或者说多中心化，无需集中的控制而能达成共识，实现上尽量分布式；
- 通过密码学的机制来确保交易无法被抵赖和破坏，并尽量保护用户信息和记录的隐私性。

分类

根据参与者的不同，可以分为公有（Public）链、联盟（Consortium）链和私有（Private）链。

公有链，顾名思义，任何人都可以参与使用和维护，典型的如比特币区块链，信息是完全公开的。

如果进一步引入许可机制，可以实现私有链和联盟链两种类型。

私有链，由集中管理者进行管理限制，只有内部少数人可以使用，信息不公开。

联盟链则介于两者之间，由若干组织一起合作维护一条区块链，该区块链的使用必须是带有权限的限制访问，相关信息会得到保护，典型如供应链机构或银行联盟。

目前来看，公有链更容易吸引市场和媒体的眼球，但更多的商业价值会在联盟链和私有链上落地。

根据使用目的和场景的不同，又可以分为以数字货币为目的的货币链，以记录产权为目的的产权链，以众筹为目的的众筹链等......，也有不局限特定应用场景的通用链。

现有大部分区块链实现都至少包括了网络层、共识层、智能合约和应用层等结构，联盟链实现往往还会引入一定的权限管理机制。

关键问题和挑战

从技术角度讲，区块链所涉及到的领域比较繁杂，包括分布式系统、存储、密码学、心理学、经济学、博弈论、控制论、网络协议等，这也就意味着大量工程实践上的技术挑战。

下面列出了目前业内关注较多的一些技术话题。

抗抵赖与隐私保护

怎么防止交易记录被篡改？

怎么证明交易双方的身份？

怎么保护交易双方的隐私？

密码学的发展，为解决这些问题提供了不少手段。传统方案包括 Hash 算法，加解密算法，数字证书和签名（盲签名、环签名）等。

随着区块链技术的应用，新出现的需求将刺激密码学的进一步发展，包括更高效的随机数产生、更高强度的加密、更快速的加解密处理等。同时，量子计算等新技术的出现，也会带来更多的挑战，例如，RSA 算法等目前商用的加密算法，在未来可能无法提供足够的安全性。

能否满足这些新的需求，将依赖于数学科学的进一步发展和新一代计算技术的突破。

分布式共识

这是个经典的技术难题，学术界和业界都已有大量的研究成果（包括 Paxos、拜占庭系列算法等）。

问题的核心在于如何解决某个变更在分布式网络中得到一致的执行结果，是被参与多方都承认的，同时这个信息是被确定的，不可推翻的。

该问题在公开匿名场景下和带权限管理的场景下需求差异较大，从而导致了基于概率的算法和确定性算法两类思想。

最初，比特币区块链考虑的是公开匿名场景下的最坏保证。通过引入了“工作量证明”（Proof of Work）策略来规避少数人的恶意行为，并通过概率模型保证最后参与方共识到最长链。算法在核心思想上是基于经济利益的博弈，让恶意破坏的参与者损失经济利益，从而保证大部分人的合作。同时，确认必须经过多个区块的生成之后达成，从概率上进行保证。这类算法的主要问题在于效率的低下。类似算法，还有以权益为抵押的 PoS、DPoS 和 Casper 等。

后来更多的区块链技术（如超级账本）在带权限管理的场景下，开始考虑支持更多的确定性的共识机制，包括经典的拜占庭算法等，可以解决快速确认的问题。

共识问题在很长一段时间内都将是极具学术价值的研究热点，核心的指标将包括容错的节点比例、决策收敛速度、出错后的恢复、动态特性等。PoW 等基于概率的系列算法理论上允许少于一半的不合作节点，PBFT 等确定性算法理论上则允许不超过 1/3 的不合作节点。

交易性能

虽然一般说来，区块链不适用于高频交易的场景，但由于金融系统的需求，业界目前十分关心如何提高区块链系统交易的吞吐量，同时降低交易的确认延迟。

目前，公开的比特币区块链只能支持平均每秒约 7 笔的吞吐量，一般认为对于大额交易来说，安全的交易确认时间为一个小时左右。以太坊区块链的吞吐量略高一些，但交易性能也被认为是较大的瓶颈。

注：实际上，小额交易只要确认被广播到网络中并带有合适的交易服务费用，即有较大概率被最终打包。

区块链系统跟传统分布式系统不同，其处理性能很难通过单纯增加节点数来进行横向扩展。实际上，传统区块链系统的性能，在很大程度上取决于单个节点的处理能力。高性能、安全、稳定性、硬件辅助加解密能力，都将是考察节点性能的核心要素。

这种场景下，为了提高处理性能，一方面可以提升单个节点的性能（如采用高配置的硬件），同时设计优化的策略和算法，提高性能；另外一方面试图将大量高频的交易放到链外来，只用区块链记录最终交易信息，如比特币社区提出的 闪电网络 等设计。类似地，侧链（side chain）、影子链（shadow chain）等思路在当前阶段也有一定的借鉴意义。类似设计可以将交易性能提升 1~2 个数量级。

此外，在联盟链的场景下，参与多方存在一定的信任前提和利益约束，可以采取更优化的设计，换来性能的提升。以超级账本 Fabric 项目为例，在普通虚拟机配置下，单客户端每秒可以达到数百次（Transactions per second，tps）的交易吞吐量；在有一定工程优化或硬件加速情况下可以达到每秒数千次的吞吐量。

客观地说，目前开源区块链系统已经可以满足不少应用场景的性能需求，但离大规模交易系统的每秒数万笔的吞吐性能还有较大差距。

注：据公开的数据，*VISA* 系统的处理均值为 *2,000 tps*，峰值为 *56,000 tps*；某金融支付系统的处理峰值超过了 *85,000 tps*；某大型证券交易所号称的处理均（峰）值在 *80,000 tps* 左右。

扩展性

常见的分布式系统，可以通过横向增加节点来扩展整个系统的处理能力。

对于区块链网络系统来说，根据共识机制的不同，这个问题往往并非那么简单。

例如，对于比特币和以太坊区块链而言，网络中每个参与维护的核心节点都要保持一份完整的存储，并且进行智能合约的处理。此时，整个网络的总存储和计算能力，取决于单个节点的能力。甚至当网络中节点数过多时，可能会因为一致性的达成过程延迟降低整个网络的性能。尤其在公有网络中，由于大量低性能处理节点的存在，问题将更加明显。

要解决这个问题，根本上是放松对每个节点都必须参与完整处理的限制（当然，网络中节点要能合作完成完整的处理），这个思路已经在超级账本中得到应用；同时尽量减少核心层的处理工作。

在联盟链模式下，还可以专门采用高性能的节点作为核心节点，用相对较弱的节点仅作为代理访问节点。

安全防护

区块链目前最热门的应用场景是金融相关的服务，安全自然是讨论最多、挑战最大的话题。

区块链在设计上大量采用了现代成熟的密码学算法。但这是否就能确保其绝对安全呢？

世界上并没有绝对安全的系统。

系统是由人设计的，系统也是由人来运营的，只要有人参与的系统，就难免出现漏洞。

有如下几个方面是很难逃避的。

首先是立法。对区块链系统如何进行监管？攻击区块链系统是否属于犯罪？攻击银行系统是要承担后果的。但是目前还没有任何法律保护区块链（特别是公有链）以及基于它的实现。

其次是软件实现的潜在漏洞是无法避免的。考虑到使用了几十年的 openssl 还带着那么低级的漏洞（heart bleeding），而且是源代码完全开放的情况下，让人不禁对运行中的大量线上系统持谨慎态度。而对于金融系统来说，无论客户端还是平台侧，即便是很小的漏洞都可能造成难以估计的损失。

另外，公有区块链所有交易记录都是公开可见的，这意味着所有的交易，即便被匿名化和加密处理，但总会在未来某天被破解。安全界一般认为，只要物理上可接触就不是彻底的安全。实际上，已有文献证明，比特币区块链的交易记录很大可能是能追踪到真实用户的。

作为一套完全分布式的系统，公有的区块链缺乏有效的调整机制。一旦运行起来，出现问题也难以修正。即使是让它变得更高效、更完善的修改，只要有部分既得利益者联合起来反对，就无法得到实施。比特币社区已经出现过多次类似的争论。

最后，运行在区块链上的智能合约应用可能是五花八门的，可能存在潜在的漏洞，必须要有办法进行安全管控，在注册和运行前需要有合理的机制进行探测，以规避恶意代码的破坏。

2016 年 6 月 17 日，发生 DAO 系统漏洞被利用 事件，直接导致价值 6000 万美元的数字货币被利用者获取。尽管对于这件事情的反思还在进行中，但事实再次证明，目前基于区块链技术进行生产应用时，务必要细心谨慎地进行设计和验证。必要时，甚至要引入“形式化验

证”和人工审核机制。

可以参考，著名黑客米特尼克所著的《反欺骗的艺术——世界传奇黑客的经历分享》，介绍了大量的实际社交工程欺骗场景。

数据库和存储系统

区块链网络中的大量信息需要写到文件和数据库中进行存储。

观察区块链的应用，大量的读写操作、Hash 计算和验证操作，跟传统数据库的行为十分不同。

当年，人们观察到互联网应用大量非事务性的查询操作，而设计了非关系型（NoSQL）数据库。那么，针对区块链应用的这些特点，是否可以设计出一些特殊的针对性的数据库呢？

LevelDB、RocksDB 等键值数据库，具备很高的随机写和顺序读、写性能，以及相对较差的随机读的性能，被广泛应用到了区块链信息存储中。但目前来看，面向区块链的数据库技术仍然是需要突破的技术难点之一，特别是如何支持更丰富语义的操作。

大胆预测，未来将可能出现更具针对性的“块数据库（BlockDB）”，专门服务类似区块链这样的新型数据业务，其中每条记录将包括一个完整的区块信息，并天然地跟历史信息进行关联，一旦写入确认则无法修改。所有操作的最小单位将是一个块。为了实现这种结构，需要原生支持高效的签名和加解密处理。

集成和运营

即便大量企业系统准备迁移到区块链平台上，在相当长的一段时间内，基于区块链的新业务系统必将与已有的中心化系统集成共存。

两种系统如何共存，如何分工，彼此的业务交易如何进行合理传递？出现故障如何排查和隔离？已有数据如何在不同系统之间进行迁移和灾备？区块链系统自身又该如何进行运营，包括网络的设计选择、状态监控、灾备等。

这些都是很迫切要解决的实际问题。解决不好，将是区块链技术落地的不小阻碍。

趋势与展望

关于区块链技术发展趋势的探讨和争论，自其诞生之日起就从未停息。或许，读者可以从计算技术的演变历史中得到一些启发。

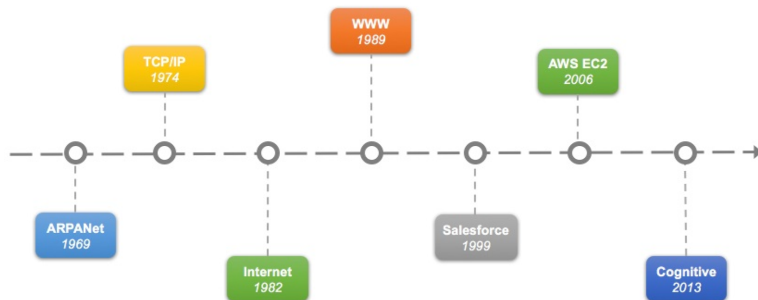


图 *1.5.4.1* - 计算的历史，笔者于某次技术交流会中提出

以云计算为代表的现代计算技术，其发展历史上有若干重要的时间点和事件：

- 1969 - ARPANet（Advanced Research Projects Agency Network）：现代互联网的前身，被美国高级研究计划署（Advanced Research Project Agency）提出，其使用 NCP 协议，核心缺陷之一是无法做到和个别计算机网络交流；
- 1973 - TCP/IP：Vinton.Cerf（文特•瑟夫）与 Bob Karn（鲍勃•卡恩）共同开发出 TCP 模型，解决了 NCP 的缺陷；
- 1982 - Internet：TCP/IP 正式成为规范，并被大规模应用，现代互联网诞生；
- 1989 - WWW：早期互联网的应用主要包括 telnet、ftp、email 等，蒂姆·伯纳斯-李（Tim Berners-Lee）设计的 WWW 协议成为互联网的杀手级应用，引爆了现代互联网，从那开始，互联网业务快速扩张；
- 1999 - salesforce：互联网出现后，一度只能进行通信应用，但 salesforce 开始以云的理念提供基于互联网的企业级服务；
- 2006 - aws ec2：AWS EC2 奠定了云计算的业界标杆，直到今天，竞争者们仍然在试图追赶 AWS 的脚步；
- 2013 - cognitive：以 IBM Watson 为代表的认知计算开始进入商业领域，计算开始变得智能，进入“后云计算时代”。

从这个历史中能看出哪些端倪呢？

一个是 技术领域也存在着周期律。 这个周期目前看是 7-8 年左右。或许正如人有“七年之痒”，技术也存在着七年这道坎，到了这道坎，要么自身突破迈过去，要么就被新的技术所取代。如果从比特币网络上线（2009 年 1 月）算起，到今年正是在坎上。因此，现在正是相关技术进行突破的好时机。

为何恰好是七年？*7* 年按照产品周期来看基本是 *2~3* 个产品周期，所谓事不过三，经过 *2~3* 个产品周期也差不多该有个结论了。

另外，最早出现的未必是先驱，也可能是先烈。 创新固然很好，但过早播撒的种子，没有合适的土壤，往往也难长大。技术创新与科研创新很不同的一点便是，技术创新必须立足于需求，过早过晚都会错失良机。科研创新则要越早越好，比如二十世纪出现的物理学巨匠们，超前的研究成果奠定了后续一百多年的科技革命的基础。

最后，事物的发展往往是延续的、长期的。 新生事物大都不是凭空蹦出来的，往往是解决了前辈未能解决的问题，或是出现了之前未曾出现过的场景。而且很多时候，新生事物的出现需要长期的孵化，坚持还是放弃，故事不断重复。笔者认为，只要是朝着提高生产力的正确方向努力，迟早会有出现在舞台上的一天。

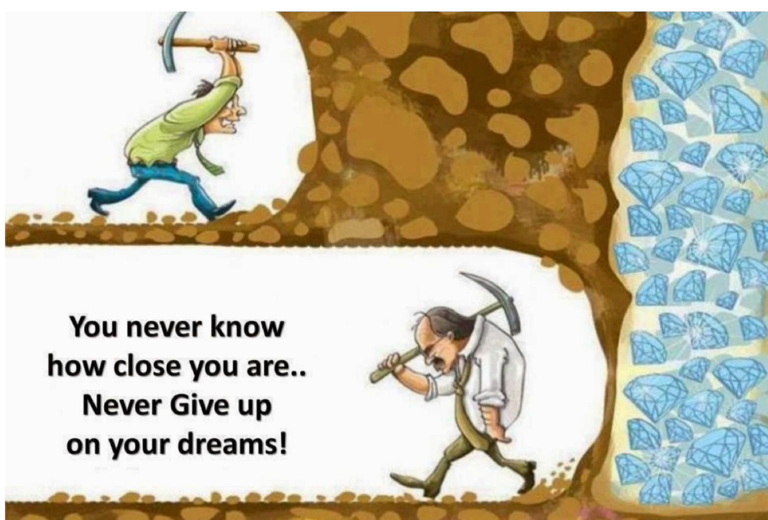


图 *1.5.4.2* - 坚持还是放弃？

目前，区块链在数字货币领域（以比特币为代表）的应用已经相对成熟，而在智能合约和分布式账本方向尚处于初步实践阶段。但毫无疑问地是，区块链技术在已经落地的领域，确实带来了生产力提升。

国际上的标准化组织，包括 ITU-T (SG16,17,20)、IETF、IEEE (P2418)、ISO、W3C 都已经纷纷设立相关研究小组，制定分布式账本领域的相关标准和规范。这也意味着分布式账本技术已经得到了业界的认可和重视。

认识上的误区

目前，由于区块链自身仍是一种相对年轻的技术，不少人对区块链的认识还存在一些误区。下面，是需要注意的一些问题：

首先，区块链不等于比特币。虽说区块链的基本思想诞生于比特币的设计中，但发展到今日，比特币和区块链已经俨然成为了两个不太相关的技术。前者更侧重从数字货币角度发掘比特币的实验性意义；后者则从技术层面探讨和研究可能带来的商业系统价值，试图在更多的场景下释放智能合约和分布式账本带来的科技潜力。

其次，区块链不等于数据库。虽然区块链也可以用来存储数据，但它要解决的核心问题是多方的互信问题。单纯从存储数据角度，它的效率可能不高，笔者也不推荐把大量的原始数据放到区块链系统上。当然，现在已有的区块链系统中，数据库相关的技术十分关键，直接决定了区块链系统的吞吐性能。

TODO: bigchaindb, chainsql

最后，区块链并非一门万能的颠覆性技术。作为融合多项已有技术而出现的新事物，区块链跟现有技术的关系是一脉相承的。它在解决多方合作和可信处理上向前多走了一步，但并不意味着它是万能的，更不会彻底颠覆已有的商业模式。很长一段时间里，区块链所适用的场景仍需不断摸索，并且跟已有系统也必然是长期合作共存的关系。

本章小结

本章剖析了区块链的相关核心技术，包括其定义、工作原理、技术分类、关键问题和认识上的误区等。通过本章的学习，读者可以对区块链的相关核心技术形成整体上的认识，并对区块链在整个信息科技产业中的位置和发展趋势形成更清晰的认知。

除了数字货币应用外，现在业界越来越看重区块链技术可能带来的面向商业应用场景的计算能力。开源社区发起的开放的 以太坊 和 超级账本 等项目，让用户可以使用它们来快速设计更复杂的分布式账本应用。

有理由相信，随着更多商业应用场景的出现，区块链技术将在未来金融和信息技术等领域占据越来越重要的地位。

典型应用场景

科技创新，应用为王。

一项新技术能否最终落地普及，有很多影响因素。其中很关键的一点便是能否找到合适的应用场景。

以比特币网络为代表的大规模数字货币系统，长时间自治运行，支持了传统金融系统都难以实现的全球范围即时可靠交易。这为区块链技术的应用潜力引发了无限遐想。如果未来基于区块链技术构造的商业价值网络成为现实，所有的交易都将高效完成且无法伪造；所有签署的合同都能按照约定严格执行。这将极大降低整个商业体系运转的成本，同时大大提高社会沟通协作的效率。从这个意义上讲，基于区块链技术构建的未来商业网络，将可能引发继互联网之后又一次巨大的产业变革。

目前，金融交易系统已经开始验证和使用区块链系统。包括征信管理、跨国交易、跨组织合作、资源共享和物联网等诸多领域，也涌现出大量有趣的应用案例。本章将通过剖析这些典型的应用场景，展现区块链技术为不同行业带来的创新潜力。

应用场景概览

区块链技术已经从单纯的技术探讨走向了应用落地的阶段。国内外已经出现大量与之相关的企业和团队。有些企业已经结合自身业务摸索出了颇具特色的应用场景，更多的企业还处于不断探索和验证的阶段。

实际上，要找到合适的应用场景，还是要从区块链技术自身的特性出发进行分析。

区块链在不引入第三方中介机构的前提下，可以提供去中心化、不可篡改、安全可靠等特性保证。因此，所有直接或间接依赖于第三方担保机构的活动，均可能从区块链技术中获益。

区块链自身维护着一个按时间顺序持续增长、不可篡改的数据记录，当现实或数字世界中的资产可以生成数字摘要时，区块链便成为确权类应用的完美载体，提供包含所属权和时间戳的数字证据。

可编程的智能合约使得在区块链上登记的资产可以获得在现实世界中难以提供的流动性，并能够保证合约规则的透明和不可篡改。这就为区块链上诞生更多创新的经济活动提供了土壤，为社会资源价值提供更加高效且安全的流动渠道。

此外，还需要思考区块链解决方案的合理边界。面向大众消费者的区块链应用需要做到公开、透明、可审计，既可以部署在无边界的公有链，也可以部署在应用生态内多中心节点共同维护的区块链；面向企业内部或多个企业间的商业区块链场景，则可将区块链的维护节点和可见性限制在联盟内部，并用智能合约重点解决联盟成员间信任或信息不对等问题，以提高经济活动效率。

笔者认为，未来几年内，可能深入应用区块链技术的场景将包括：

- 金融服务：区块链带来的潜在优势包括降低交易成本、减少跨组织交易风险等。该领域的区块链应用目前最受关注，全球不少银行和金融交易机构都是主力推动者。部分投资机构也在应用区块链技术降低管理成本和管控风险。从另一方面，要注意可能引发的问题和风险。例如，DAO（Decentralized Autonomous Organization 是史上最大的一次众筹活动，基于区块链技术确保资金的管理和投放）这样的众筹实验，提醒应用者在业务和运营层面都要谨慎处理。
- 征信和权属管理：征信和权属的数字化管理是大型社交平台和保险公司都梦寐以求的。目前该领域的主要技术问题包括缺乏足够的数据和分析能力；缺乏可靠的平台支持以及有效的数据整合管理等。区块链被认为可以促进数据交易和流动，提供安全可靠的支持。征信行业的门槛比较高，需要多方资源共同推动。
- 资源共享：以 Airbnb 为代表的分享经济公司将欢迎去中心化应用，可以降低管理成本。该领域主题相对集中，设计空间大，受到大量的投资关注。
- 贸易管理：区块链技术可以帮助自动化国际贸易和物流供应链领域中繁琐的手续和流程。基于区块链设计的贸易管理方案会为参与的多方企业带来极大的便利。另外，贸易中销售和法律合同的数字化、货物监控与检测、实时支付等方向都可能成为创业公司的

突破口。

- 物联网：物联网也是很适合应用区块链技术的一个领域，预计未来几年内会有大量应用出现，特别是租赁、物流等特定场景，都是很合适结合区块链技术的场景。但目前阶段，物联网自身的技术局限将造成短期内不会出现大规模应用。

这些行业各有不同的特点，但或多或少都需要第三方担保机构的参与，因此都可能从区块链技术中获得益处。

当然，对于商业系统来说，技术支持只是一种手段，根本上需要满足业务需求。区块链作为一个底层的平台技术，要利用好它，需要根据行业特性进行综合考量设计，对其上的业务系统和商业体系提供合理的支持。

有理由相信，区块链技术落地的案例会越来越多。这也会进一步促进新技术在传统行业中的应用，带来更多的创新业务和场景。

金融服务

自有人类社会以来，金融交易就是必不可少的经济活动，涉及货币、证券、保险、抵押、捐赠等诸多行业。交易角色和交易功能的不同，反映出不同的生产关系。通过金融交易，可以优化社会运转效率，实现资源价值的最大化。可以说，人类社会的文明发展，离不开交易形式的演变。

传统交易本质上交换的是物品价值的所属权。为了完成一些贵重商品的交易（例如房屋、车辆的所属权），往往需要十分繁琐的中间环节，同时需要中介和担保机构参与其中。这是因为，交易双方往往存在着不能充分互信的情况。一方面，要证实合法的价值所属权并不简单，往往需要开具各种证明材料，存在造假的可能；另一方面，价值不能直接进行交换，同样需要繁琐的手续，在这个过程中存在较多的篡改风险。

为了确保金融交易的可靠完成，出现了中介和担保机构这样的经济角色。它们通过提供信任保障服务，提高了社会经济活动的效率。但现有的第三方中介机制往往存在成本高、时间周期长、流程复杂、容易出错等缺点。金融领域长期存在提高交易效率的迫切需求。

区块链技术可以为金融服务提供有效、可信的所属权证明，以及相当可靠的合约确保机制。

银行业金融管理

银行从角色上，一般分为中央银行（央行）和普通银行。

中央银行的两大职能是“促进宏观经济稳定”和“维护金融稳定”（《金融的本质》，伯克南，中信出版社，2014 年出版），主要手段就是管理各种证券和利率。央行的存在，为整个社会的金融体系提供了最终的信用担保。

普通银行业则往往基于央行的信用，作为中介和担保方，来协助完成多方的金融交易。

银行活动主要包括发行货币、完成存贷款等功能。银行必须确保交易的确定性，必须确立自身的可靠信用地位。传统的金融系统为了完成上述功能，采用了极为复杂的软件和硬件方案，建设和维护成本都十分昂贵。即便如此，这些系统仍然存在诸多缺陷，例如某些场景下交易时延过大；难以避免利用系统漏洞进行的攻击和金融欺诈等。

此外，在目前金融系统流程中，商家为了完成交易，还常常需要经由额外的支付企业进行处理。这些实际上都极大增加了现有金融交易的成本。

区块链技术的出现，被认为是有可能促使这一行业发生革命性变化的“奇点”。除了众所周知的比特币等数字货币实验之外，还有诸多金融机构进行了有意义的尝试。

欧洲央行评估区块链在证券交易后结算的应用

目前，全球证券交易后的处理过程十分复杂，清算行为成本约 50-100 亿美元，交易后分析、对账和处理费用超过 200 亿美金。

来自欧洲央行的一份报告显示，区块链作为分布式账本技术，可以很好地节约对账的成本，同时简化交易过程。相对原先的交易过程，可以近乎实时的变更证券的所有权。

来源：欧洲央行报告：《*Distributed ledger technologies in securities post-trading*》，*https://www.ecb.europa.eu/pub/pdf/scpops/ecbop172.en.pdf*

中国人民银行投入区块链研究

2016 年，中国人民银行对外发布消息，称深入研究了数字货币涉及的相关技术，包括区块链技术、移动支付、可信可控云计算、密码算法、安全芯片等，被认为积极关注区块链技术的发展。

实际上，央行对于区块链技术的研究很早便已开展。

2014 年，央行成立发行数字货币的专门研究小组对基于区块链的数字货币进行研究，次年形成研究报告。

2016 年 1 月 20 日，央行专门组织了“数字货币研讨会”，邀请了业内的区块链技术专家就数字货币发行的总体框架、演进、以及国家加密货币等话题进行了研讨。会后，发布对我国银行业数字货币的战略性发展思路，提出要早日发行数字货币，并利用数字货币相关技术来打击金融犯罪活动。

2016 年 12 月，央行成立数字货币研究所。初步公开设计为“由央行主导，在保持实物现金发行的同时发行以加密算法为基础的数字货币，M0（流通中的现金）的一部分由数字货币构成。为充分保障数字货币的安全性，发行者可采用安全芯片为载体来保护密钥和算法运算过程的安全”。

加拿大银行提出新的数字货币

2016 年 6 月，加拿大央行公开正在开发基于区块链技术的数字版加拿大元（名称为 CAD 币），以允许用户使用加元来兑换该数字货币。经过验证的对手方将会处理货币交易；另外，如果需要，银行将保留销毁 CAD 币的权利。

发行 CAD 币是更大的一个探索型科技项目 Jasper 的一部分。除了加拿大央行外，据悉，蒙特利尔银行、加拿大帝国商业银行、加拿大皇家银行、加拿大丰业银行、多伦多道明银行等多家机构也都参与了该项目。

来源：金融时报- *Canada experiments with digital dollar on blockchain*，*2016-06-16*。

英国央行实现 **RSCoin**

英国央行在数字化货币方面进展十分突出，已经实现了基于分布式账本平台的数字化货币系统，RSCoin。旨在强化本国经济及国际贸易。

RSCoin 目标是提供一个由中央银行控制的数字货币，采用了双层链架构、改进版的两阶段提交（Two Phase Commitment），以及多链之间的交叉验证机制。该货币具备防篡改和防伪造的特性。

因为该系统主要是央行和下属银行之间使用，通过提前建立一定的信任基础，可以提供较好的处理性能。

英国央行对 RSCoin 进行了推广，希望能尽快普及该数字货币，以带来节约经济成本、促进经济发展的效果。同时，英国央行认为，数字货币相对传统货币更适合国际贸易等场景。

日本政府取消比特币消费税

2017 年 3 月 27 日，日本国会通过《2017 税务改革法案》，该法案将比特币等数字货币定义为货币等价物，可以用于数字支付和转账。

法案于 2017 年 7 月 1 日生效，销售数字货币不必再缴纳 8% 的消费税。

中国邮储银行将区块链技术应用到核心业务系统

2016 年 10 月，中国邮储银行宣布携手 IBM 推出基于区块链技术的资产托管系统，是中国银行业首次将区块链技术成功应用于核心业务系统。

新的业务系统免去了重复的信用校验过程，将原有业务环节缩短了约 60-80% 的时间，提高了信用交易的效率。

SWIFT 完成跨银行的分布式账本验证

2018 年 3 月，环球同业银行金融电讯协会（SWIFT)完成了涉及到 34 家银行的分布式账本验证。验证重点关注基于超级账本项目的分布式账本技术能否满足监管、安全、隐私性等方面的需求。验证表明分布式账本技术可以满足自动化的资产管理需求，为未来多银行间合作提供重要支撑。

SWIFT 研发中心负责人 Damien Vanderveken 称："验证进行的相当不错，证实了分布式账本技术的巨大进展，尤其是超级账本 Fabric 项目 1.0（The PoC went extremely well, proving the fantastic progress that has been made with DLT and the Hyperledger Fabric 1.0 in particular）"。

各种新型支付业务

基于区块链技术，出现了大量的创新支付企业，这些支付企业展示了利用区块链技术带来的巨大商业优势。

- Abra：区块链数字钱包，以近乎实时的速度进行跨境支付，无需银行账户，实现不同币种的兑换，融资超过千万美金。
- Bitwage：基于比特币区块链的跨境工资支付平台，可以实现每小时的工资支付，方便跨国企业进行工资管理。
- BitPOS：澳大利亚创业企业，提供基于比特币的低成本的快捷线上支付。
- Circle：由区块链充当支付网络，允许用户进行跨币种、跨境的快速汇款。Circle 获得了来自 IDG、百度的超过 6000 万美金的 D 轮投资。
- Ripple：实现跨境的多币种、低成本、实时交易，引入了网关概念（类似银行），结构偏中心化。

证券交易

证券交易包括交易执行环节和交易后处理环节。

交易环节本身相对简单，主要是由交易系统（高性能实时处理系统）完成电子数据库中内容的变更。中心化的验证系统往往极为复杂和昂贵。交易指令执行后的结算和清算环节也十分复杂，需要大量的人力成本和时间成本，并且容易出错。

目前来看，基于区块链的处理系统还难以实现海量交易系统所需要的性能（典型性能为每秒一万笔以上成交，日处理能力超过五千万笔委托、三千万笔成交）。但在交易的审核和清算环节，区块链技术存在诸多的优势，可以极大降低处理时间，同时减少人工的参与。

> 咨询公司 Oliver Wyman 在给 SWIFT（环球同业银行金融电讯协会）提供的研究报告中预计，全球清算行为成本约 50~100 亿美元，结算成本、托管成本和担保物管理成本 400~450 亿美元（390 亿美元为托管链的市场主体成本），而交易后流程数据及分析花费 200~250 亿美元。

2015 年 10 月，美国纳斯达克（Nasdaq）证券交易所推出区块链平台 Nasdaq Linq，实现主要面向一级市场的股票交易流程。通过该平台进行股票发行的发行者将享有“数字化”的所有权。

其它相关案例还包括：

- BitShare 推出基于区块链的证券发行平台，号称每秒达到 10 万笔交易。
- DAH 为金融市场交易提供基于区块链的交易系统。获得澳洲证交所项目。
- Symbiont 帮助金融企业创建存储于区块链的智能债券，当条件符合时，清算立即执行。
- Overstock.com 推出基于区块链的私有和公开股权交易“T0”平台，提出“交易即结算”（The trade is the settlement）的理念，主要目标是建立证券交易实时清算结算的全新系统。
- 高盛为一种叫做“SETLcoin”的新虚拟货币申请专利，用于为股票和债券等资产交易提供“近乎立即执行和结算”的服务。

众筹管理

ICO（Initial Coin Offering）是一种基于加密货币的新型众筹管理方式。项目发起方通过售卖项目早期的加密货币向外界融资，当项目上线后，如果能否得以健康成长，加密货币价格上溢，投资者可以获得回报，并且可以选择任何时候卖出这些货币而退出。

最早的 ICO 出现在 2013 年 6 月，万事达币(MSC)在 Bitcointalk 论坛上众筹 5000 个比特币。虽然，很可惜该项目后来并没有成功，但开启了 ICO 的浪潮。

2014 年，比较出名的如比特股 Bitshares 和以太坊 Ethereum 先后发起 ICO，并且随着平台自身的发展，投资者获取了大量的回报。

2016 年 4 月 30 日上线的 DAO（Decentralized Autonomous Organization）项目，试图打造基于以太坊的众筹平台，更是一度创下历史最高的融资记录，数额超过 1.6 亿美金。该项目暴露出这种创新形式的组织者们在应对安全风险时候缺乏足够的应对经验。6 月 12 日，有技术人员报告合约执行过程中存在软件漏洞，但很遗憾并未得到组织的重视和及时修复。四天后，黑客利用漏洞转移了 360 万枚以太币，当时价值超过 5000 万美金。虽然最后采用了一些技术手段来挽回经济损失，但该事件毫无疑问给以太坊平台带来了负面影响，也给 ICO 这种新模式的流程管理敲响了警钟。

2017 年开始，传统风投基金也开始尝试用 ICO 来募集资金。Blockchain Capital 在 2017 年发行的一支基金创新地采用了传统方式加 ICO 的混合方式进行募资，其中传统部分规模 4000 万美元，ICO 部分规模 1000 万美元。4 月 10 日，ICO 部分 1000 万美元的募集目标在启动后六小时内全部完成。

随着 ICO 的火热发展，大量欺诈性的项目出现，有的项目仅有一个粗糙的白皮书，有的项目甚至连白皮书都没有，在部分媒体和投资者的炒作下，吸引了众多资金的投入，2017 年全球有超过 40 亿美金投入到 ICO 当中。然而，下半年开始，大量 ICO 项目因为无法完成预设目标而破灭，无论欧美还是亚洲都出现了大量项目组织方跑路的情况，这被认为是第一次 ICO 泡沫的结束。

同期，各国监管部门开始介入，要么直接禁止 ICO 活动，要么纳入监管体系。2017年 8 月 28 日，美国证监会发布关于谨防 ICO 骗局的警告；9 月 4 日，中国人民银行（央行）等7部门发文定调，ICO 为“未经批准非法公开融资的行为”，各类代币发行融资活动应立即停止。

客观来看，用 ICO 方式进行众筹相对灵活。但目前 ICO 在形式还缺少明确的法律定位和监管流程，普通投资者也很难对项目的商业模式和科技含量进行准确把握，是一件风险远大于收益的事情。

Telegram 在 2018 年初通过两轮 ICO 共募集资金 17 亿美金，值得注意的是，在第二轮时已经明确限制最低投资门槛为 100 万美元。

ICO 形式要想健康发展，一方面必须强化项目自身的管理，如信息的真实公开和接受公开机构的监督审查；另一方面，投资者必须从商业场景、技术实力、团队等各方面对项目进行全面评估，而不是盲目跟从。

征信和权属管理

征信管理

征信管理是一个巨大的潜在市场，据称超过千亿规模（可参考美国富国银行报告和平安证券报告），也是目前大数据应用领域最有前途的方向之一。

目前征信相关的大量有效数据集中在少数机构手中。由于这些数据太过敏感，并且具备极高的商业价值，往往会被严密保护起来，形成很高的行业门槛。

虽然现在大量的互联网企业（包括各类社交网站）尝试从各种维度获取了海量的用户信息，但从征信角度看，这些数据仍然存在若干问题。这些问题主要包括：

- 数据量不足：数据量越大，能获得的价值自然越高，过少的数据量无法产生有效价值；
- 相关度较差：最核心的数据也往往是最敏感的。在隐私高度敏感的今天，用户都不希望暴露过多数据给第三方，因此企业获取到数据中有效成分往往很少；
- 时效性不足：企业可以从明面上获取到的用户数据往往是过时的，甚至存在虚假信息，对相关分析的可信度造成严重干扰。

区块链天然存在着无法篡改、不可抵赖的特性。同时，区块链平台将可能提供前所未有规模的相关性极高的数据，这些数据可以在时空中准确定位，并严格关联到用户。因此，基于区块链提供数据进行征信管理，将大大提高信用评估的准确率，同时降低评估成本。

另外，跟传统依靠人工的审核过程不同，区块链中交易处理完全遵循约定自动化执行。基于区块链的信用机制将天然具备稳定性和中立性。

目前，包括 IDG、腾讯、安永、普华永道等都已投资或进入基于区块链的征信管理领域，特别是跟保险和互助经济相关的应用场景。

权属管理

区块链技术可以用于产权、版权等所有权的管理和追踪。其中包括汽车、房屋、艺术品等各种贵重物品的交易等，也包括数字出版物，以及可以标记的数字资源。

目前权属管理领域存在的几个难题是：

- 物品所有权的确认和管理；
- 交易的安全性和可靠性保障；
- 必要的隐私保护机制。

以房屋交易为例。买卖双方往往需要依托中介机构来确保交易的进行，并通过纸质的材料证明房屋所有权。但实际上，很多时候中介机构也无法确保交易的正常进行。

而利用区块链技术，物品的所有权是写在数字链上的，谁都无法修改。并且一旦出现合同中约定情况，区块链技术将确保合同能得到准确执行。这能有效减少传统情况下纠纷仲裁环节的人工干预和执行成本。

例如，公正通（Factom）尝试使用区块链技术来革新商业社会和政府部门的数据管理和数据记录方式。包括审计系统、医疗信息记录、供应链管理、投票系统、财产契据、法律应用、金融系统等。它将待确权数据的指纹存放到基于区块链的分布式账本中，可以提供资产所有权的追踪服务。

区块链账本共享、信息可追踪溯源且不可篡改的特性同样可用于打击造假和防范欺诈。Everledger 自 2016 年起就研究基于区块链技术实现贵重资产检测系统，将钻石或者艺术品等的权属信息记录在区块链上。并于 2017 年宣布与 IBM 合作，实现生产商、加工商、运送方、零售商等多方之间的可信高效协作。

类似地，针对食品造假这一难题，IBM、沃尔玛、清华大学于 2016 年底共同宣布将在食品安全领域展开合作，将用区块链技术搭建透明可追溯的跨境食品供应链。这一全新的供应链将改善食品的溯源和物流环节，打造更为安全的全球食品市场。

其他项目

在人力资源和教育领域，MIT 研究员朱莉安娜·纳扎雷（Juliana Nazaré）和学术创新部主管菲利普·施密特（Philipp Schmidt）发表了文章《MIT Media Lab Uses the Bitcoin Blockchain for Digital Certificates》，介绍基于区块链的学历认证系统。基于该系统，用人单位可以确认求职者的学历信息是真实可靠的。2018 年 2 月，麻省理工学院向应届毕业生颁发了首批基于区块链的数字学位证书。

此外，还包括一些其他相关的应用项目：

- Chronicled：基于区块链的球鞋鉴定方案，为正品球鞋添加电子标签，记录在区块链上。
- Mediachain：通过 metadata 协议，将内容创造者与作品唯一对应。
- Mycelia：区块链产权保护项目，为音乐人实现音乐的自由交易。
- Tierion: 将用户数据锚定在比特币或以太坊区块链上，并生成“区块链收据”。
- Ziggurat：基于区块链提供文字、图片、音视频版权资产的登记和管理服务。

资源共享

当前，以 Uber、Airbnb 为代表的共享经济模式正在多个垂直领域冲击传统行业。这一模式鼓励人们通过互联网的方式共享闲置资源。资源共享目前面临的问题主要包括：

- 共享过程成本过高；
- 用户行为评价难；
- 共享服务管理难。

区块链技术为解决上述问题提供了更多可能。相比于依赖于中间方的资源共享模式，基于区块链的模式有潜力更直接的连接资源的供给方和需求方，其透明、不可篡改的特性有助于减小摩擦。

有人认为区块链技术会成为新一代共享经济的基石。笔者认为，区块链在资源共享领域是否存在价值，还要看能否比传统的专业供应者或中间方形式实现更高的效率和更低的成本，同时不能损害用户体验。

短租共享

大量提供短租服务的公司已经开始尝试用区块链来解决共享中的难题。

高盛在报告《Blockchain: Putting Theory into Practice》中宣称：

> Airbnb 等 P2P 住宿平台已经开始通过利用私人住所打造公开市场来变革住宿行业，但是这种服务的接受程度可能会因人们对人身安全以及财产损失的担忧而受到限制。而如果通过引入安全且无法篡改的数字化资质和信用管理系统，我们认为区块链就能有助于提升 P2P 住宿的接受度。

该报告还指出，可能采用区块链技术的企业包括 Airbnb、HomeAway 以及 OneFineStay 等，市场规模为 30~90 亿美元。

社区能源共享

在纽约布鲁克林的一个街区，已有项目尝试将家庭太阳能发的电通过社区的电力网络直接进行买卖。具体的交易不再经过电网公司，而是通过区块链执行。

与之类似，ConsenSys 和微电网开发商 LO3 提出共建光伏发电交易网络，实现点对点的能源交易。

这些方案的主要难题包括：

- 太阳能电池管理；
- 社区电网构建；

* 电力储备系统搭建；
* 低成本交易系统支持。

现在已经有大量创业团队在解决这些问题，特别是硬件部分已经有了不少解决方案。而通过区块链技术打造的平台可以解决最后一个问题，即低成本地实现社区内的可靠交易系统。

电商平台

传统情况下，电商平台起到了中介的作用。一旦买卖双方发生纠纷，电商平台会作为第三方机构进行仲裁。这种模式存在着周期长、缺乏公证、成本高等缺点。OpenBazaar 试图在无中介的情形下，实现安全电商交易。

具体地，OpenBazaar 提供的分布式电商平台，通过多方签名机制和信誉评分机制，让众多参与者合作进行评估，实现零成本解决纠纷问题。

大数据共享

大数据时代里，价值来自于对数据的挖掘，数据维度越多，体积越大，潜在价值也就越高。

一直以来，比较让人头疼的问题是如何评估数据的价值，如何利用数据进行交换和交易，以及如何避免宝贵的数据在未经许可的情况下泄露出去。

区块链技术为解决这些问题提供了潜在的可能。

利用共同记录的共享账本，数据在多方之间的流动将得到实时的追踪和管理。通过对敏感信息的脱敏处理和访问权限的设定，区块链可以对大数据的共享授权进行精细化管控，规范和促进大数据的交易与流通。

减小共享风险

传统的资源共享平台在遇到经济纠纷时会充当调解和仲裁者的角色。对于区块链共享平台，目前还存在线下复杂交易难以数字化等问题。除了引入信誉评分、多方评估等机制，也有方案提出引入保险机制来对冲风险。

2016 年 7 月，德勤、Stratumn 和 LemonWay 共同推出一个为共享经济场景设计的“微保险”概念平台，称为 LenderBot。针对共享经济活动中临时交换资产可能产生的风险，LenderBot 允许用户在区块链上注册定制的微保险，并为共享的资产（如相机、手机、电脑）投保。区块链在其中扮演了可信第三方和条款执行者的角色。

贸易管理

跨境贸易

在国际贸易活动中，买卖双方可能互不信任。因此需要银行作为买卖双方的保证人，代为收款交单，并以银行信用代替商业信用。

区块链可以为信用证交易参与方提供共同账本，允许银行和其它参与方拥有经过确认的共同交易记录并据此履约，从而降低风险和成本。

巴克莱银行用区块链进行国际贸易结算

2016 年 9 月，英国巴克莱银行用区块链技术完成了一笔国际贸易的结算，贸易金额 10 万美元，出口商品是爱尔兰农场出产的芝士和黄油，进口商是位于离岸群岛塞舌尔的一家贸易商。结算用时不到 4 小时，而传统采用信用证方式做此类结算需要 7 到 10 天。

在这笔贸易背后，区块链提供了记账和交易处理系统，替代了传统信用证结算过程中占用大量人力和时间的审单、制单、电报或邮寄等流程。

物流供应链

物流供应链被认为是区块链一个很有前景的应用方向。

供应链行业往往涉及到诸多实体，包括物流、资金流、信息流等，这些实体之间存在大量复杂的协作和沟通。传统模式下，不同实体各自保存各自的供应链信息，严重缺乏透明度，造成了较高的时间成本和金钱成本，而且一旦出现问题（冒领、货物假冒等），难以追查和处理。

通过区块链，各方可以获得一个透明可靠的统一信息平台，可以实时查看状态，降低物流成本，追溯物品的生产和运送全过程，从而提高供应链管理的效率。当发生纠纷时，举证和追查也变得更加清晰和容易。

例如，运送方通过扫描二维码来证明货物到达指定区域，并自动收取提前约定的费用；冷链运输过程中通过温度传感器实时检测货物的温度信息并记录在链等。

来自美国加州的 Skuchain 公司创建基于区块链的新型供应链解决方案，实现商品流与资金流的同步，同时缓解假货问题。

马士基推出基于区块链的跨境供应链解决方案

2017 年 3 月，马士基和 IBM 宣布，计划与由货运公司、货运代理商、海运承运商、港口和海关当局构成的物流网络合作构建一个新型全球贸易数字化解决方案。该方案利用区块链技术在各方之间实现信息透明性，降低贸易成本和复杂性，旨在帮助企业减少欺诈和错误，缩短产品在运输和海运过程中所花的时间，改善库存管理，最终减少浪费并降低成本。

马士基在 2014 年发现，仅仅是将冷冻货物从东非运到欧洲，就需要经过近 30 个人员和组织进行超过 200 次的沟通和交流。类似这样的问题都有望借助区块链进行解决。

一带一路

类似“一带一路”这样创新的投资建设模式，会碰到来自地域、货币、信任等各方面的挑战。

现在已经有一些参与到一带一路中的部门，对区块链技术进行探索应用。区块链技术可以让原先无法交易的双方（例如，不存在多方都认可的国际货币储备的情况下）顺利完成交易，并且降低贸易风险、减少流程管控的成本。

物联网

曾经有人认为，物联网是大数据时代的基础。

笔者认为，区块链技术是物联网时代的基础。

典型应用场景分析

一种可能的应用场景为：物联网络中每一个设备分配地址，给该地址所关联一个账户，用户通过向账户中支付费用可以租借设备，以执行相关动作，从而达到租借物联网的应用。

典型的应用包括 PM2.5 监测点的数据获取、温度检测服务、服务器租赁、网络摄像头数据调用等等。

另外，随着物联网设备的增多、边沿计算需求的增强，大量设备之间形成分布式自组织的管理模式，并且对容错性要求很高。区块链自身分布式和抗攻击的特点可以很好地融合到这一场景中。

IBM

IBM 在物联网领域已经持续投入了几十年的研发，目前正在探索使用区块链技术来降低物联网应用的成本。

2015 年初，IBM 与三星宣布合作研发“去中心化的 P2P 自动遥测系统（Autonomous Decentralized Peer-to-Peer Telemetry）”系统，使用区块链作为物联网设备的共享账本，打造去中心化的物联网。

Filament

美国的 Filament 公司以区块链为基础提出了一套去中心化的物联网软件堆栈。通过创建一个智能设备目录，Filament 的物联网设备可以进行安全沟通、执行智能合约以及发送小额交易。

基于上述技术，Filament 能够通过远程无线网络将辽阔范围内的工业基础设备沟通起来，其应用包括追踪自动售货机的存货和机器状态、检测铁轨的损耗、基于安全帽或救生衣的应急情况监测等。

NeuroMesh

2017 年 2 月，源自 MIT 的 NeuroMesh 物联网安全平台获得了 MIT 100K Accelerate 竞赛的亚军。该平台致力于成为“物联网疫苗”，能够检测和消除物联网中的有害程序，并将攻击源打入黑名单。

所有运行 NeuroMesh 软件的物联网设备都通过访问区块链账本来识别其他节点和辨认潜在威胁。如果一个设备借助深度学习功能检测出可能的威胁，可通过发起投票的形式告知全网，由网络进一步对该威胁进行检测并做出处理。

公共网络服务

现有的互联网能正常运行，离不开很多近乎免费的网络服务，例如域名服务（DNS）。任何人都可以免费查询到域名，没有 DNS，现在的各种网站将无法访问。因此，对于网络系统来说，类似的基础服务必须要能做到安全可靠，并且低成本。

区块链技术恰好具备这些特点，基于区块链打造的分布式 DNS 系统，将减少错误的记录和查询，并且可以更加稳定可靠地提供服务。

其它场景

区块链还有一些很有趣的应用场景，包括但不限于云存储、医疗、社交、游戏等多个方面。

云存储

Storj 项目提供了基于区块链的安全的分布式云存储服务。服务保证只有用户自己能看到自己的数据，并号称提供高速的下载速度和 99.99999% 的高可用性。用户还可以“出租”自己的额外硬盘空间来获得报酬。

协议设计上，Storj 网络中的节点可以传递数据、验证远端数据的完整性和可用性、复原数据，以及商议合约和向其他节点付费。数据的安全性由数据分片（Data Sharding）和端到端加密提供，数据的完整性由可复原性证明（Proof of Retrievability）提供。

医疗

医院与医保医药公司，不同医院之间，甚至医院里不同部门之间的数据流动性往往很差。考虑到医疗健康数据的敏感性，笔者认为，如果能够满足数据访问权、使用权等规定的基础上促进医疗数据的提取和流动，区块链将在医疗行业获得一定的用武之地。

GemHealth 项目由区块链公司 Gem 于 2016 年 4 月提出，其目标除了用区块链存储医疗记录或数据，还包括借助区块链增强医疗健康数据在不同机构不同部门间的安全可转移性、促进全球病人身份识别、医疗设备数据安全收集与验证等。项目已与医疗行业多家公司签订了合作协议。

通信和社交

BitMessage 是一套去中心化通信系统，在点对点通信的基础上保护用户的匿名性和信息的隐私。BitMessage 协议在设计上充分参考了比特币，二者拥有相似的地址编码机制和消息传递机制。BitMessage 也用工作量证明（Proof-of-Work）机制防止通信网络受到大量垃圾信息的冲击。

类似的，Twister 是一套去中心化的“微博”系统，Dot-Bit 是一套去中心化的 DNS 系统。

投票

Follow My Vote 项目致力于提供一个安全、透明的在线投票系统。通过使用该系统进行选举投票，投票者可以随时检查自己选票的存在和正确性，看到实时记票结果，并在改变主意时修改选票。

该项目使用区块链进行记票，并开源其软件代码供社区用户审核。项目也为投票人身份认证、防止重复投票、投票隐私等难点问题提供了解决方案。

预测

Augur 是一个运行在以太坊上的预测市场平台。使用 Augur，来自全球不同地方的任何人都可发起自己的预测话题市场，或随意加入其它市场，来预测一些事件的发展结果。预测结果和奖金结算由智能合约严格控制，使得在平台上博弈的用户不用为安全性产生担忧。

电子游戏

2017 年 3 月，来自马来西亚的电子游戏工作室 Xhai Studios 宣布将区块链技术引入其电子游戏平台。工作室旗下的一些游戏将支持与 NEM 区块链的代币 XEM 整合。通过这一平台，游戏开发者可以在游戏架构中直接调用支付功能，消除对第三方支付的依赖；玩家则可以自由地将 XEM 和游戏内货币、点数等进行双向兑换。

本章小结

本章介绍了大量基于区块链技术的应用案例和场景，展现了区块链以及基于区块链的分布式账本技术所具有的巨大市场潜力。

当然，任何事物的发展都不是一帆风顺的。

目前来看，制约区块链技术进一步落地的因素有很多。比如如何来为区块链上的合同担保？特别在金融、法律等领域，实际执行的时候往往还需要线下机制来配合；另外就是基于区块链系统的价值交易，必须要实现物品价值的数字化，非数字化的物品很难直接放到数字世界中进行管理。

这些问题看起来都不容易很快得到解决。但笔者相信，一门新的技术能否站住脚，根本上还是看它能否最终提高生产力，而区块链技术已经证明了这一点。随着生态的进一步成熟，区块链技术必将在更多领域获得用武之地。

分布式系统

万法皆空，因果不空。

随着摩尔定律碰到瓶颈，越来越多的系统要依靠分布式集群架构来实现海量数据处理和可扩展计算能力。

区块链首先是一个分布式系统。

中央式结构改成分布式系统，碰到的第一个问题就是一致性的保障。

很显然，如果一个分布式集群无法保证处理结果一致的话，那任何建立于其上的业务系统都无法正常工作。

本章将介绍分布式系统中一些核心问题的来源以及相关的工作。

一致性问题

在分布式系统中，一致性(Consistency，早期也叫 Agreement)是指对于系统中的多个服务节点，给定一系列操作，在协议（往往通过某种共识算法）保障下，试图使得它们对处理结果达成某种程度的一致。

如果分布式系统能实现“一致”，对外就可以呈现为一个功能正常的，且性能和稳定性都要好很多的“虚处理节点”。

举个例子，某影视公司旗下有西单和中关村的两个电影院，都出售某电影票，票一共就一万张。那么，顾客到达某个电影院买票的时候，售票员该怎么决策是否该卖这张票，才能避免超售呢？当电影院个数更多的时候呢？

这个问题在人类世界中，看起来似乎没那么难，你看，英国人不是刚靠 投票 达成了“某种一致”吗？

注意：一致性并不代表结果正确与否，而是系统对外呈现的状态一致与否，例如，所有节点都达成失败状态也是一种一致。

挑战

在实际的计算机集群系统（看似强大的计算机系统，很多地方都比人类世界要脆弱的多）中，存在如下的问题：

- 节点之间的网络通讯是不可靠的，包括任意延迟和内容故障；
- 节点的处理可能是错误的，甚至节点自身随时可能宕机；
- 同步调用会让系统变得不具备可扩展性。

要解决这些挑战，愿意动脑筋的读者可能会很快想出一些不错的思路。

为了简化理解，仍然以两个电影院一起卖票的例子。可能有如下的解决思路：

- 每次要卖一张票前打电话给另外一家电影院，确认下当前票数并没超售；
- 两家电影院提前约好，奇数小时内一家可以卖票，偶数小时内另外一家可以卖；
- 成立一个第三方的存票机构，票都放到他那里，每次卖票找他询问；
- 更多……

这些思路大致都是可行的。实际上，这些方法背后的思想，将可能引发不一致的并行操作进行串行化，就是现在计算机系统里处理分布式一致性问题的基础思路和唯一秘诀。只是因为计算机系统比较傻，需要考虑得更全面一些；而人们又希望计算机系统能工作的更快更稳定，所以算法需要设计得再精巧一些。

要求

规范的说，理想的分布式系统一致性应该满足：

- 可终止性（Termination）：一致的结果在有限时间内能完成；
- 共识性（Consensus）：不同节点最终完成决策的结果应该相同；
- 合法性（Validity）：决策的结果必须是其它进程提出的提案。

第一点很容易理解，这是计算机系统可以被使用的前提。需要注意，在现实生活中这点并不是总能得到保障的，例如取款机有时候会是“服务中断”状态，电话有时候是“无法连通”的。

第二点看似容易，但是隐藏了一些潜在信息。算法考虑的是任意的情形，凡事一旦推广到任意情形，就往往有一些惊人的结果。例如现在就剩一张票了，中关村和西单的电影院也分别刚确认过这张票的存在，然后两个电影院同时来了一个顾客要买票，从各自“观察”看来，自己的顾客都是第一个到的……怎么能达成结果的共识呢？记住我们的唯一秘诀：核心在于需要把两件事情进行排序，而且这个顺序还得是大家都认可的。

第三点看似绕口，但是其实比较容易理解，即达成的结果必须是节点执行操作的结果。仍以卖票为例，如果两个影院各自卖出去一千张，那么达成的结果就是还剩八千张，决不能认为票售光了。

带约束的一致性

做过分布式系统的读者应该能意识到，绝对理想的强一致性（Strong Consistency）代价很大。除非不发生任何故障，所有节点之间的通信无需任何时间，这个时候其实就等价于一台机器了。实际上，越强的一致性要求往往意味着越弱的性能。

一般的，强一致性（Strong Consistency）主要包括下面两类：

- 顺序一致性（Sequential Consistency）：Leslie Lamport 1979 年经典论文《How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs》中提出，是一种比较强的约束，保证所有进程看到的 全局执行顺序（total order）一致，并且每个进程看自身的执行（local order）跟实际发生顺序一致。例如，某进程先执行 A，后执行 B，则实际得到的全局结果中就应该为 A 在 B 前面，而不能反过来。同时所有其它进程在全局上也应该看到这个顺序。顺序一致性实际上限制了各进程内指令的偏序关系，但不在进程间按照物理时间进行全局排序。
- 线性一致性（Linearizability Consistency）：Maurice P. Herlihy 与 Jeannette M. Wing 在 1990 年经典论文《Linearizability: A Correctness Condition for Concurrent Objects》中共同提出，在顺序一致性前提下加强了进程间的操作排序，形成唯一的全局顺序（系统等价于是顺序执行，所有进程看到的所有操作的序列顺序都一致，并且跟实际发生顺序一致），是很强的原子性保证。但是比较难实现，目前基本上要么依赖于全局的时钟或锁，要么通过一些复杂算法实现，性能往往不高。

目前，高精度的石英钟的漂移率为 10^{-7}，人类目前最准确的原子震荡时钟的漂移率为 10^{-13}。Google 曾在其分布式数据库 Spanner 中采用基于原子时钟和 GPS 的“TrueTime”方案，能够将不同数据中心的时间偏差控制在 10ms 以内。方案简单粗暴而有效，但存在成本较高的

问题。

强一致的系统往往比较难实现。很多时候，人们发现实际需求并没有那么强，可以适当放宽一致性要求，降低系统实现的难度。例如在一定约束下实现所谓最终一致性（Eventual Consistency），即总会存在一个时刻（而不是立刻），系统达到一致的状态，这对于大部分的 Web 系统来说已经足够了。这一类弱化的一致性，被笼统称为弱一致性（Weak Consistency）。

莫非分布式领域也有一个测不准原理？这个世界为何会有这么多的约束呢？

共识算法

实际上，要保障系统满足不同程度的一致性，往往需要通过共识算法来达成。

共识算法解决的是对某个提案（Proposal），大家达成一致意见的过程。提案的含义在分布式系统中十分宽泛，如多个事件发生的顺序、某个键对应的值、谁是领导......等等，可以认为任何需要达成一致的信息都是一个提案。

注：实践中，一致性的结果往往还需要客户端的特殊支持，典型地通过访问足够多个服务节点来验证确保获取共识后结果。

问题挑战

实际上，如果分布式系统中各个节点都能保证以十分强大的性能（瞬间响应、高吞吐）无故障的运行，则实现共识过程并不复杂，简单通过多播过程投票即可。

很可惜的是，现实中这样“完美”的系统并不存在，如响应请求往往存在时延、网络会发生中断、节点会发生故障、甚至存在恶意节点故意要破坏系统。

一般地，把故障（不响应）的情况称为“非拜占庭错误”，恶意响应的情况称为“拜占庭错误”（对应节点为拜占庭节点）。

常见算法

针对非拜占庭错误的情况，一般包括 Paxos、Raft 及其变种。

对于要能容忍拜占庭错误的情况，一般包括 PBFT 系列、PoW 系列算法等。从概率角度，PBFT 系列算法是确定的，一旦达成共识就不可逆转；而 PoW 系列算法则是不确定的，随着时间推移，被推翻的概率越来越小。

理论界限

搞学术的人都喜欢对问题先确定一个界限，那么，这个问题的最坏界限在哪里呢？很不幸，一般情况下，分布式系统的共识问题无解。

当节点之间的通信网络自身不可靠情况下，很显然，无法确保实现共识。但好在，一个设计得当的网络可以在大概率上实现可靠的通信。

然而，即便在网络通信可靠情况下，一个可扩展的分布式系统的共识问题的下限是无解。

这个结论，被称为 `FLP 不可能性` 原理，可以看做分布式领域的“测不准原理”。

FLP 不可能性原理

> FLP 不可能原理：在网络可靠，存在节点失效（即便只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。

提出该定理的论文是由 Fischer, Lynch 和 Patterson 三位作者于 1985 年发表，该论文后来获得了 Dijkstra（就是发明最短路径算法的那位）奖。

FLP 不可能原理实际上告诉人们，不要浪费时间去为异步分布式系统设计在任意场景下都能实现共识的算法。

理解这一原理的一个不严谨的例子是：

三个人在不同房间，进行投票（投票结果是 0 或者 1）。三个人彼此可以通过电话进行沟通，但经常会有人时不时地睡着。比如某个时候，A 投票 0，B 投票 1，C 收到了两人的投票，然后 C 睡着了。A 和 B 则永远无法在有限时间内获知最终的结果。如果可以重新投票，则类似情形每次在取得结果前发生：（

FLP 原理实际上说明对于允许节点失效情况下，纯粹异步系统无法确保一致性在有限时间内完成。

这岂不是意味着研究一致性问题压根没有意义吗？

先别这么悲观，学术界做研究，考虑的是数学和物理意义上最极端的情形，很多时候现实生活要美好的多（感谢这个世界如此鲁棒！）。例如，上面例子中描述的最坏情形，总会发生的概率并没有那么大。工程实现上多试几次，很大可能就成功了。

科学告诉你什么是不可能的；工程则告诉你，付出一些代价，我可以把它变成可能。

这就是工程的魅力。

那么，退一步讲，在付出一些代价的情况下，我们能做到多少？

回答这一问题的是另一个很出名的原理：CAP 原理。

科学上告诉你去赌场赌博从概率上总会是输钱的；工程则告诉你，如果你愿意接受最终输钱的结果，中间说不定偶尔能小赢几笔呢！？

CAP 原理

CAP 原理最早由 Eric Brewer 在 2000 年，ACM 组织的一个研讨会上提出猜想，后来 Lynch 等人进行了证明。

该原理被认为是分布式系统领域的重要原理。

定义

分布式计算系统不可能同时确保一致性（Consistency）、可用性（Availablity）和分区容忍性（Partition），设计中往往需要弱化对某个特性的保证。

- 一致性（Consistency）：任何操作应该都是原子的，发生在后面的事件能看到前面事件发生导致的结果，注意这里指的是强一致性；
- 可用性（Availablity）：在有限时间内，任何非失败节点都能应答请求；
- 分区容忍性（Partition）：网络可能发生分区，即节点之间的通信不可保障。

比较直观地理解，当网络可能出现分区时候，系统是无法同时保证一致性和可用性的。要么，节点收到请求后因为没有得到其他人的确认就不应答，要么节点只能应答非一致的结果。

好在大部分时候网络被认为是可靠的，因此系统可以提供一致可靠的服务；当网络不可靠时，系统要么牺牲掉一致性（大部分时候都是如此），要么牺牲掉可用性。

应用场景

既然 CAP 不可同时满足，则设计系统时候必然要弱化对某个特性的支持。

弱化一致性

对结果一致性不敏感的应用，可以允许在新版本上线后过一段时间才更新成功，期间不保证一致性。

例如网站静态页面内容、实时性较弱的查询类数据库等，CouchDB、Cassandra 等为此设计。

弱化可用性

对结果一致性很敏感的应用，例如银行取款机，当系统故障时候会拒绝服务。MongoDB、Redis 等为此设计。

Paxos、Raft 等算法，主要处理这种情况。

弱化分区容忍性

现实中，网络分区出现概率减小，但较难避免。某些关系型数据库、ZooKeeper 即为此设计。

实践中，网络通过双通道等机制增强可靠性，达到高稳定的网络通信。

ACID 原则

即 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）。

ACID 原则描述了对分布式数据库的一致性需求，同时付出了可用性的代价。

- Atomicity：每次操作是原子的，要么成功，要么不执行；
- Consistency：数据库的状态是一致的，无中间状态；
- Isolation：各种操作彼此互相不影响；
- Durability：状态的改变是持久的，不会失效。

一个与之相对的原则是 BASE（Basic Availiability，Soft state，Eventually Consistency），牺牲掉对一致性的约束（最终一致性），来换取一定的可用性。

Paxos 与 Raft

Paxos 问题是指分布式的系统中存在故障（fault），但不存在恶意（corrupt）节点场景（即可能消息丢失或重复，但无错误消息）下的共识达成（Consensus）问题。因为最早是 Leslie Lamport 用 Paxon 岛的故事模型来进行描述而命名。

Paxos

1990 年由 Leslie Lamport 提出的 Paxos 共识算法，在工程角度实现了一种最大化保障分布式系统一致性（存在极小的概率无法实现一致）的机制。Paxos 被广泛应用在 Chubby、ZooKeeper 这样的系统中，Leslie Lamport 因此获得了 2013 年度图灵奖。

故事背景是古希腊 Paxon 岛上的多个法官在一个大厅内对一个议案进行表决，如何达成统一的结果。他们之间通过服务人员来传递纸条，但法官可能离开或进入大厅，服务人员可能偷懒去睡觉。

Paxos 是第一个被证明的共识算法，其原理基于 两阶段提交 并进行扩展。

作为现在共识算法设计的鼻祖，以最初论文的难懂（算法本身并不复杂）出名。算法中将节点分为三种类型：

- proposer：提出一个提案，等待大家批准为结案。往往是客户端担任该角色；
- acceptor：负责对提案进行投票。往往是服务端担任该角色；
- learner：被告知结案结果，并与之统一，不参与投票过程。可能为客户端或服务端。

并且，算法需要满足 safety 和 liveness 两方面的约束要求（实际上这两个基础属性是大部分分布式算法都该考虑的）：

- safety：保证决议结果是对的，无歧义的，不会出现错误情况。
    - 决议（value）只有在被 proposers 提出的 proposal 才能被最终批准；
    - 在一次执行实例中，只批准（chosen）一个最终决议，意味着多数接受（accept）的结果能成为决议；
- liveness：保证决议过程能在有限时间内完成。
    - 决议总会产生，并且 learners 能获得被批准（chosen）的决议。

基本过程包括 proposer 提出提案，先争取大多数 acceptor 的支持，超过一半支持时，则发送结案结果给所有人进行确认。一个潜在的问题是 proposer 在此过程中出现故障，可以通过超时机制来解决。极为凑巧的情况下，每次新的一轮提案的 proposer 都恰好故障，系统则永远无法达成一致（概率很小）。

Paxos 能保证在超过 $1/2$ 的正常节点存在时，系统能达成共识。

读者可以试着自己设计一套能达成共识的方案，会发现在满足各种约束情况下，算法自然就会那样设计。

单个提案者+多接收者

如果系统中限定只有某个特定节点是提案者，那么一致性肯定能达成（只有一个方案，要么达成，要么失败）。提案者只要收到了来自多数接收者的投票，即可认为通过，因为系统中不存在其他的提案。

但一旦提案者故障，则系统无法工作。

多个提案者+单个接收者

限定某个节点作为接收者。这种情况下，共识也很容易达成，接收者收到多个提案，选第一个提案作为决议，拒绝掉后续的提案即可。

缺陷也是容易发生单点故障，包括接收者故障或首个提案者节点故障。

以上两种情形其实类似主从模式，虽然不那么可靠，但因为原理简单而被广泛采用。

当提案者和接收者都推广到多个的情形，会出现一些挑战。

多个提案者+多个接收者

既然限定单提案者或单接收者都会出现故障，那么就得允许出现多个提案者和多个接收者。问题一下子变得复杂了。

一种情况是同一时间片段（如一个提案周期）内只有一个提案者，这时可以退化到单提案者的情形。需要设计一种机制来保障提案者的正确产生，例如按照时间、序列、或者大家猜拳（出一个数字来比较）之类。考虑到分布式系统要处理的工作量很大，这个过程要尽量高效，满足这一条件的机制非常难设计。

另一种情况是允许同一时间片段内可以出现多个提案者。那同一个节点可能收到多份提案，怎么对他们进行区分呢？这个时候采用只接受第一个提案而拒绝后续提案的方法也不适用。很自然的，提案需要带上不同的序号。节点需要根据提案序号来判断接受哪个。比如接受其中序号较大（往往意味着是接受新提出的，因为旧提案者故障概率更大）的提案。

如何为提案分配序号呢？一种可能方案是每个节点的提案数字区间彼此隔离开，互相不冲突。为了满足递增的需求可以配合用时间戳作为前缀字段。

此外，提案者即便收到了多数接收者的投票，也不敢说就一定通过。因为在此过程中系统可能还有其它的提案。

两阶段的提交

提案者发出提案之后，收到一些反馈。一种结果是自己的提案被大多数接受了，另一种结果是没被接受。没被接受的话好说，过会再试试。

即便收到来自大多数的接受反馈，也不能认为就最终确认了。因为这些接收者自己并不知道自己刚反馈的提案就恰好是全局的绝大多数。

很自然的，引入了新的一个阶段，即提案者在前一阶段拿到所有的反馈后，判断这个提案是可能被大多数接受的提案，需要对其进行最终确认。

Paxos 里面对这两个阶段分别命名为准备（prepare）和提交（commit）。准备阶段解决大家对哪个提案进行投票的问题，提交阶段解决确认最终值的问题。

准备阶段：

- 提案者发送自己计划提交的提案的编号到多个接受者，试探是否可以锁定多数接受者的支持。
- 接受者时刻保留收到过提案的最大编号和接受的最大提案。如果收到的提案号比目前保留的最大提案号还大，则返回自己已接受的提案值（如果还未接受过任何提案，则为空）给提案者，更新当前最大提案号，并说明不再接受小于最大提案号的提案。

提交阶段：

- 提案者如果收到大多数的回复（表示大部分人听到它的请求），则可准备发出带有刚才提案号的接受消息。如果收到的回复中不带有新的提案，说明锁定成功，则使用自己的提案内容；如果返回中有提案内容，则替换提案值为返回中编号最大的提案值。如果没收到足够多的回复，则需要再次发出请求。
- 接受者收到接受消息后，如果发现提案号不小于已接受的最大提案号，则接受该提案，并更新接受的最大提案。

一旦多数接受了共同的提案值，则形成决议，成为最终确认的提案。

Raft

Raft 算法是Paxos 算法的一种简化实现。

包括三种角色：leader、candidate 和 follower，其基本过程为：

- Leader 选举：每个 candidate 随机经过一定时间都会提出选举方案，最近阶段中得票最多者被选为 leader；
- 同步 log：leader 会找到系统中 log 最新的记录，并强制所有的 follower 来刷新到这个记录；

注：此处 *log* 并非是指日志消息，而是各种事件的发生记录。

拜占庭问题与算法

拜占庭问题更为广泛，讨论的是允许存在少数节点作恶（消息可能被伪造）场景下的一致性达成问题。拜占庭算法讨论的是最坏情况下的保障。

中国将军问题

拜占庭将军问题之前，就已经存在中国将军问题：两个将军要通过信使来达成进攻还是撤退的约定，但信使可能迷路或被敌军阻拦（消息丢失或伪造），如何达成一致。根据 FLP 不可能原理，这个问题无解。

拜占庭问题

又叫拜占庭将军（Byzantine Generals Problem）问题，是 Leslie Lamport 1982 年提出用来解释一致性问题的一个虚构模型。拜占庭是古代东罗马帝国的首都，由于地域宽广，守卫边境的多个将军（系统中的多个节点）需要通过信使来传递消息，达成某些一致的决定。但由于将军中可能存在叛徒（系统中节点出错），这些叛徒将努力向不同的将军发送不同的消息，试图会干扰一致性的达成。

拜占庭问题即为在此情况下，如何让忠诚的将军们能达成行动的一致。

对于拜占庭问题来说，假如节点总数为 N，叛变将军数为 F，则当 时，问题才有解，即 Byzantine Fault Tolerant (BFT) 算法。

例如，$N=3, F=1$ 时。

提案人不是叛变者，提案人发送一个提案出来，叛变者可以宣称收到的是相反的命令。则对于第三个人（忠诚者）收到两个相反的消息，无法判断谁是叛变者，则系统无法达到一致。

提案人是叛变者，发送两个相反的提案分别给另外两人，另外两人都收到两个相反的消息，无法判断究竟谁是叛变者，则系统无法达到一致。

更一般的，当提案人不是叛变者，提案人提出提案信息 1，则对于合作者来看，系统中会有 $N-F$ 份确定的信息 1，和 F 份不确定的信息（可能为 0 或 1，假设叛变者会尽量干扰一致的达成），$N-F>F$，即 $N>2F$ 情况下才能达成一致。

当提案人是叛变者，会尽量发送相反的提案给 $N-F$ 个合作者，从收到 1 的合作者看来，系统中会存在 个信息 1，以及 个信息 0 ；从收到 0 的合作者看来，系统中会存在 个信息 0 ，以及 个信息 1 ；

另外存在 $F-1$ 个不确定的信息。合作者要想达成一致，必须进一步的对所获得的消息进行判定，询问其他人某个被怀疑对象的消息值，并通过取多数来作为被怀疑者的信息值。这个过程可以进一步递归下去。

Leslie Lamport 证明，当叛变者不超过 时，存在有效的算法，不论叛变者如何折腾，忠诚的将军们总能达成一致的结果。如果叛变者过多，则无法保证一定能达到一致性。

多于 的叛变者时有没有可能有解决方案呢？设想 f 个叛变者和 g 个忠诚者，叛变者故意使坏，可以给出错误的结果，也可以不响应。某个时候 f 个叛变者都不响应，则 g 个忠诚者取多数既能得到正确结果。当 f 个叛变者都给出一个恶意的提案，并且 g 个忠诚者中有 f 个离线时，剩下的 $g-f$ 个忠诚者此时无法分别是否混入了叛变者，仍然要确保取多数能得到正确结果，因此，$g-f>f$，即 $g>2f$，所以系统整体规模要大于 $3f$。

能确保达成一致的拜占庭系统节点数至少为 4，允许出现 1 个坏的节点。

Byzantine Fault Tolerant 算法

面向拜占庭问题的容错算法，解决的是网络通信可靠，但节点可能故障情况下的一致性达成。

最早由 Castro 和 Liskov 在 1999 年提出的 Practical Byzantine Fault Tolerant（PBFT）是第一个得到广泛应用的 BFT 算法。只要系统中有 的节点是正常工作的，则可以保证一致性。

PBFT 算法包括三个阶段来达成共识：Pre-Prepare、Prepare 和 Commit。

新的解决思路

拜占庭问题之所以难解，在于任何时候系统中都可能存在多个提案（因为提案成本很低），并且要完成最终的一致性确认过程十分困难，容易受干扰。但是一旦确认，即为最终确认。

比特币的区块链网络在设计时提出了创新的 PoW（Proof of Work） 算法思路。一个是限制一段时间内整个网络中出现提案的个数（增加提案成本），另外一个是放宽对最终一致性确认的需求，约定好大家都确认并沿着已知最长的链进行拓宽。系统的最终确认是概率意义上的存在。这样，即便有人试图恶意破坏，也会付出很大的经济代价（付出超过系统一半的算力）。

后来的各种 PoX 系列算法，也都是沿着这个思路进行改进，采用经济上的惩罚来制约破坏者。

可靠性指标

很多领域一般都喜欢谈服务可靠性，用几个 9 来说事。这几个 9 其实是粗略代表了概率意义上系统能提供服务的可靠性指标，最初是电信领域提出的概念。

下表给出不同指标下，每年允许服务出现不可用时间的参考值。

指标	概率可靠性	每年允许不可用时间	典型场景
一个九	90%	1.2 个月	不可用
二个九	99%	3.6 天	普通单点
三个九	99.9%	8.6 小时	普通企业
四个九	99.99%	51.6 分钟	高可用
五个九	99.999%	5 分钟	电信级
六个九	99.9999%	31 秒	极高要求
七个九	99.99999%	3 秒	N/A
八个九	99.999999%	0.3 秒	N/A
九个九	99.9999999%	30 毫秒	N/A

一般来说，单点的服务器系统至少应能满足两个九；普通企业信息系统三个九就肯定足够了（大家可以统计下自己企业内因系统维护每年要停多少时间），系统能达到四个九已经是业界领先水平了（参考 AWS）。电信级的应用一般号称能达到五个九，这已经很厉害了，一年里面最多允许五分钟的服务停用。六个九和以上的系统，就更加少见了，要实现往往意味着极高的代价。

那么，该如何提升可靠性呢？有两个思路：一是让系统中的单点变得更可靠；二是消灭单点。

IT 从业人员大都有类似的经验，运行某软系统的机器，基本上是过几天就要重启下的；而运行 Linux 系统的服务器，则可能几年时间都不出问题。另外，普通的家用计算机，跟专用服务器相比，长时间运行更容易出现故障。这些都是单点可靠性不同的例子。可以通过替换单点的软硬件来改善可靠性。

然而，依靠单点实现的可靠性毕竟是有限的，要想进一步的提升，那就只好消灭单点，通过主从、多活等模式让多个节点集体完成原先单点的工作。这可以从概率意义上改善服务的可靠性，也是分布式系统的一个重要用途。

小结

分布式系统领域是计算机科学中十分重要的一个技术领域。

常见的分布式一致性是个古老而重要的问题，无论在学术上还是工程上都存在很高的价值。理想化（各项指标均最优）的解决方案是不存在的。

在现实各种约束条件下，往往需要通过牺牲掉某些需求，来设计出满足特定场景的协议。

其实，工程领域中很多问题的解决思路，都在于如何合理地进行取舍（trade-off）。

密码学技术

工程领域从来没有黑科技；密码学不是工程。

密码学在信息技术领域的重要地位无需多言。如果没有现代密码学的研究成果，人类社会根本无法进入信息时代。

密码学领域十分繁杂，本章将介绍密码学领域中跟区块链相关的一些基础知识，包括 hash 算法与摘要、加密算法、数字签名和证书、PKI 体系、Merkle 树、同态加密等，以及如何使用这些技术实现信息的机密性、完整性、认证性和不可抵赖性。

Hash 算法

定义

Hash（哈希或散列）算法是信息技术领域非常基础也非常重要的技术。它能任意长度的二进制值（明文）映射为较短的固定长度的二进制值（Hash 值），并且不同的明文很难映射为相同的 Hash 值。

例如计算一段话“hello blockchain world, this is yeasy@github”的 MD5 hash 值为 `89242549883a2ef85dc81b90fb606046`。

```
$ echo "hello blockchain world, this is yeasy@github"|md5
89242549883a2ef85dc81b90fb606046
```

这意味着我们只要对某文件进行 MD5 Hash 计算，得到结果为 `89242549883a2ef85dc81b90fb606046`，这就说明文件内容极大概率上就是 “hello blockchain world, this is yeasy@github”。可见，Hash 的核心思想十分类似于基于内容的编址或命名。

注：*hash* 值在应用中又被称为指纹（*fingerprint*）、摘要（*digest*）。

注：*MD5* 是一个经典的 *hash* 算法，其和 *SHA-1* 算法都已被 证明 安全性不足应用于商业场景。

一个优秀的 hash 算法，将能实现：

- 正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。
- 逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。
- 输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。
- 冲突避免：很难找到两段内容不同的明文，使得它们的 hash 值一致（发生冲突）。

冲突避免有时候又被称为“抗碰撞性”。如果给定一个明文前提下，难以找到碰撞的另一个明文，称为“弱抗碰撞性”；如果难以找到任意两个明文，发生碰撞，则称算法具有“强抗碰撞性”。

很多场景下，也要求对于任意长的输入内容，输出定长的 *hash* 结果。

流行的算法

目前流行的 Hash 算法包括 MD5、SHA-1 和 SHA-2。

MD4（RFC 1320）是 MIT 的 Ronald L. Rivest 在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。MD4 已证明不够安全。

MD5（RFC 1321）是 Rivest 于1991年对 MD4 的改进版本。它对输入仍以 512 位分组，其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备“强抗碰撞性”。

SHA （Secure Hash Algorithm）是一个 Hash 函数族，由 NIST（National Institute of Standards and Technology）于 1993 年发布第一个算法。目前知名的 SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，因此抗穷举性更好。SHA-1 设计时基于和 MD4 相同原理，并且模仿了该算法。SHA-1 已被证明不具备“强抗碰撞性”。

为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384，和 SHA-512 算法（统称为 SHA-2），跟 SHA-1 算法原理类似。SHA-3 相关算法也已被提出。

目前，一般认为 MD5 和 SHA1 已经不够安全，推荐至少使用 SHA2-256 算法。

性能

一般的，Hash 算法都是算力敏感型，意味着计算资源是瓶颈，主频越高的 CPU 进行 Hash 的速度也越快。

也有一些 Hash 算法不是算力敏感的，例如 scrypt，需要大量的内存资源，节点不能通过简单的增加更多 CPU 来获得 hash 性能的提升。

数字摘要

顾名思义，数字摘要是对数字内容进行 Hash 运算，获取唯一的摘要值来指代原始数字内容。

数字摘要是解决确保内容没被篡改过的问题（利用 Hash 函数的抗碰撞性特点）。

数字摘要是 Hash 算法最重要的一个用途。在网络上下载软件或文件时，往往同时会提供一个数字摘要值，用户下载下来原始文件可以自行进行计算，并同提供的摘要值进行比对，以确保内容没有被修改过。

加解密算法

算法类型	特点	优势	缺陷	代表算法
对称加密	加解密密钥相同或可推算	计算效率高，加密强度高	需提前共享密钥；易泄露	DES、3DES、AES、IDEA
非对称加密	加解密密钥不相关	无需提前共享密钥	计算效率低，仍存在中间人攻击可能	RSA、ElGamal、椭圆曲线系列算法

算法体系

现代加密算法的典型组件包括：加解密算法、加密密钥、解密密钥。其中，加解密算法自身是固定不变的，一般是公开可见的；密钥则往往每次不同，并且需要保护起来，一般来说，对同一种算法，密钥长度越长，则加密强度越大。

加密过程中，通过加密算法和加密密钥，对明文进行加密，获得密文。

解密过程中，通过解密算法和解密密钥，对密文进行解密，获得明文。

根据加解密的密钥是否相同，算法可以分为对称加密（symmetric cryptography，又称公共密钥加密，common-key cryptography）和非对称加密(asymmetric cryptography，又称公钥加密，public-key cryptography)。两种模式适用于不同的需求，恰好形成互补，很多时候也可以组合使用，形成混合加密机制。

并非所有加密算法的强度都可以从数学上进行证明。公认的高强度加密算法是在经过长时间各方面实践论证后，被大家所认可，不代表其不存在漏洞。但任何时候，自行发明加密算法都是一种不太明智的行为。

对称加密

顾名思义，加解密的密钥是相同的。

优点是加解密效率高（速度快，空间占用小），加密强度高。

缺点是参与多方都需要持有密钥，一旦有人泄露则安全性被破坏；另外如何在不安全通道下分发密钥也是个问题。

对称密码从实现原理上可以分为两种：分组密码和序列密码。前者将明文切分为定长数据块作为加密单位，应用最为广泛。后者则只对一个字节进行加密，且密码不断变化，只用在一些特定领域，如数字媒介的加密等。

代表算法包括 DES、3DES、AES、IDEA 等。

- DES（Data Encryption Standard）：经典的分组加密算法，1977 年由美国联邦信息处理标准（FIPS）所采用 FIPS-46-3，将 64 位明文加密为 64 位的密文，其密钥长度为 56 位 + 8 位校验。现在已经很容易被暴力破解。
- 3DES：三重 DES 操作：加密 --> 解密 --> 加密，处理过程和加密强度优于 DES，但现在也被认为不够安全。
- AES（Advanced Encryption Standard）：美国国家标准研究所（NIST）采用取代 DES 成为对称加密实现的标准，1997~2000 年 NIST 从 15 个候选算法中评选 Rijndael 算法（由比利时密码学家 Joan Daemon 和 Vincent Rijmen 发明）作为 AES，标准为 FIPS-197。AES 也是分组算法，分组长度为 128、192、256 位三种。AES 的优势在于处理速度快，整个过程可以数学化描述，目前尚未有有效的破解手段。

适用于大量数据的加解密；不能用于签名场景；需要提前分发密钥。

注：分组加密每次只能处理固定长度的明文，因此过长的内容需要采用一定模式进行加密，《实用密码学》中推荐使用 密文分组链接（*Cipher Block Chain*，*CBC*）、计数器（*Counter*，*CTR*）模式。

非对称加密

非对称加密是现代密码学历史上最为伟大的发明，可以很好的解决对称加密需要的提前分发密钥问题。

顾名思义，加密密钥和解密密钥是不同的，分别称为公钥和私钥。

公钥一般是公开的，人人可获取的，私钥一般是个人自己持有，不能被他人获取。

优点是公私钥分开，不安全通道也可使用。

缺点是加解密速度慢，一般比对称加解密算法慢两到三个数量级；同时加密强度相比对称加密要差。

非对称加密算法的安全性往往需要基于数学问题来保障，目前主要有基于大数质因子分解、离散对数、椭圆曲线等几种思路。

代表算法包括：RSA、ElGamal、椭圆曲线（Elliptic Curve Crytosystems，ECC）系列算法。

- RSA：经典的公钥算法，1978 年由 Ron Rivest、Adi Shamir、Leonard Adleman 共同提出，三人于 2002 年获得图灵奖。算法利用了对大数进行质因子分解困难的特性，但目前还没有数学证明两者难度等价，或许存在未知算法在不进行大数分解的前提下解密。
- Diffie-Hellman 密钥交换：基于离散对数无法快速求解，可以在不安全的通道上，双方协商一个公共密钥。
- ElGamal：由 Taher ElGamal 设计，利用了模运算下求离散对数困难的特性。被应用在 PGP 等安全工具中。
- 椭圆曲线算法（Elliptic curve cryptography，ECC）：现代备受关注的算法系列，基于对

椭圆曲线上特定点进行特殊乘法逆运算难以计算的特性。最早在 1985 年由 Neal Koblitz 和 Victor Miller 分别独立提出。ECC 系列算法一般被认为具备较高的安全性，但加解密计算过程往往比较费时。

一般适用于签名场景或密钥协商，不适于大量数据的加解密。

RSA 算法等已被认为不够安全，一般推荐采用椭圆曲线系列算法。

混合加密机制

即先用计算复杂度高的非对称加密协商一个临时的对称加密密钥（会话密钥，一般相对内容来说要短的多），然后双方再通过对称加密对传递的大量数据进行加解密处理。

典型的场景是现在大家常用的 HTTPS 机制。HTTPS 实际上是利用了 Transport Layer Security/Secure Socket Layer（TLS/SSL）来实现可靠的传输。TLS 为 SSL 的升级版本，目前广泛应用的为 TLS 1.0，对应到 SSL 3.1 版本。

建立安全连接的具体步骤如下：

- 客户端浏览器发送信息到服务器，包括随机数 R1，支持的加密算法类型、协议版本、压缩算法等。注意该过程为明文。
- 服务端返回信息，包括随机数 R2、选定加密算法类型、协议版本，以及服务器证书。注意该过程为明文。
- 浏览器检查带有该网站公钥的证书。该证书需要由第三方 CA 来签发，浏览器和操作系统会预置权威 CA 的根证书。如果证书被篡改作假（中间人攻击），很容易通过 CA 的证书验证出来。
- 如果证书没问题，则用证书中公钥加密随机数 R3，发送给服务器。此时，只有客户端和服务器都拥有 R1、R2 和 R3 信息，基于 R1、R2 和 R3，生成对称的会话密钥（如 AES 算法）。后续通信都通过对称加密进行保护。

数字签名

类似在纸质合同上签名确认合同内容，数字签名用于证实某数字内容的完整性（integrity）和来源（或不可抵赖，non-repudiation）。

一个典型的场景是，A 要发给 B 一个文件（一份信息），B 如何获知所得到的文件即为 A 发出的原始版本？A 先对文件进行摘要，然后用自己的私钥进行加密，将文件和加密串都发给 B。B 收到文件和加密串后，用 A 的公钥来解密加密串，得到原始的数字摘要，跟对文件进行摘要后的结果进行比对。如果一致，说明该文件确实是 A 发过来的，并且文件内容没有被修改过。

HMAC

全称是 Hash-based Message Authentication Code，即“基于 Hash 的消息认证码”。基本过程为对某个消息，利用提前共享的对称密钥和 Hash 算法进行加密处理，得到 HMAC 值。该 HMAC 值提供方可以证明自己拥有共享的对称密钥，并且消息自身可以利用 HMAC 确保未经篡改。

HMAC(K, H, Message)

其中，K 为提前共享的对称密钥，H 为提前商定的 Hash 算法（一般为公认的经典算法），Message 为要处理的消息内容。如果不知道 K 和 H，则无法根据 Message 得到准确的 HMAC 值。

HMAC 一般用于证明身份的场景，如 A、B 提前共享密钥，A 发送随机串给 B，B 对称加密处理后把 HMAC 值发给 A，A 收到了自己再重新算一遍，只要相同说明对方确实是 B。

HMAC 主要问题是需要共享密钥。当密钥可能被多方拥有的场景下，无法证明消息确实来自某人（Non-repudiation）。反之，如果采用非对称加密方式，则可以证明。

盲签名

1983 年由 David Chaum 提出。签名者在无法看到原始内容的前提下对信息进行签名。

盲签名主要是为了实现防止追踪（unlinkability），签名者无法将签名内容和结果进行对应。典型的实现包括 RSA 盲签名)。

多重签名

n 个持有人中，收集到至少 m 个（ ）的签名，即认为合法，这种签名被称为多重签名。

其中，n 是提供的公钥个数，m 是需要匹配公钥的最少的签名个数。

群签名

1991 年由 Chaum 和 van Heyst 提出。群签名属于群体密码学的一个课题。

群签名有如下几个特点：只有群中成员能够代表群体签名（群特性）；接收者可以用公钥验证群签名（验证简单性）；接收者不能知道由群体中哪个成员所签（无条件匿名保护）；发生争议时，群体中的成员或可信赖机构可以识别签名者（可追查性）。

Desmedt 和 Frankel 在 1991 年提出了基于门限的群签名实现方案。在签名时，一个具有 n 个成员的群体共用同一个公钥，签名时必须有 t 个成员参与才能产生一个合法的签名，t 称为门限或阈值。这样一个签名称为(n, t)不可抵赖群签名。

环签名

环签名由 Rivest,shamir 和 Tauman 三位密码学家在 2001 年首次提出。环签名属于一种简化的群签名。

签名者首先选定一个临时的签名者集合,集合中包括签名者自身。然后签名者利用自己的私钥和签名集合中其他人的公钥就可以独立的产生签名,而无需他人的帮助。签名者集合中的其他成员可能并不知道自己被包含在其中。

数字证书

数字证书用来证明某个公钥是谁的，并且内容是正确的。

对于非对称加密算法和数字签名来说，很重要的一点就是公钥的分发。一旦公钥被人替换（典型的如中间人攻击），则整个安全体系将被破坏掉。

怎么确保一个公钥确实是某个人的原始公钥？

这就需要数字证书机制。

顾名思义，数字证书就是像一个证书一样，证明信息的合法性。由证书认证机构（Certification Authority，CA）来签发，权威的 CA 包括 verisign 等。

数字证书内容可能包括版本、序列号、签名算法类型、签发者信息、有效期、被签发人、签发的公开密钥、**CA** 数字签名、其它信息等等，一般使用最广泛的标准为 ITU 和 ISO 联合制定的 X.509 规范。

其中，最重要的包括 `签发的公开密钥`、`CA 数字签名` 两个信息。因此，只要通过这个证书就能证明某个公钥是合法的，因为带有 CA 的数字签名。

更进一步地，怎么证明 CA 的签名合法不合法呢？

类似的，CA 的数字签名合法不合法也是通过 CA 的证书来证明的。主流操作系统和浏览器里面会提前预置一些 CA 的证书（承认这些是合法的证书），然后所有基于他们认证的签名都会自然被认为合法。

后面章节将介绍的 PKI 体系提供了一套完整的证书管理的框架。

PKI 体系

在非对称加密中，公钥则可以通过证书机制来进行保护，如何管理和分发证书则可以通过 PKI（Public Key Infrastructure）来保障。

顾名思义，PKI 体系在现代密码学应用领域处于十分基础的地位，解决了十分核心的证书管理问题。

PKI 并不代表某个特定的密码学技术和流程，PKI 是建立在公私钥基础上实现安全可靠传递消息和身份确认的一个通用框架。实现了 PKI 的平台可以安全可靠地管理网络中用户的密钥和证书，包括多个实现和变种，知名的有 RSA 公司的 PKCS（Public Key Cryptography Standards）标准和 X.509 规范等。

一般情况下，PKI 至少包括如下组件：

- CA（Certification Authority）：负责证书的颁发和吊销，接收来自 RA 的请求，是最核心的部分；
- RA（Registration Authority）：对用户身份进行验证，校验数据合法性，负责登记，审核过了就发给 CA；
- 证书数据库：存放证书，一般采用 LDAP 目录服务，标准格式采用 X.500 系列。

CA 是最核心的组件，主要完成对证书的管理。

常见的流程为，用户通过 RA 登记申请证书，CA 完成证书的制造，颁发给用户。用户需要撤销证书则向 CA 发出申请。

之前章节内容介绍过，密钥有两种类型：用于签名和用于加解密，对应称为 `签名密钥对` 和 `加密密钥对` 。

用户证书可以有两种方式。一般可以由 CA 来生成证书和私钥；也可以自己生成公钥和私钥，然后由 CA 来对公钥进行签发。后者情况下，当用户私钥丢失后，CA 无法完成恢复。

Merkle 树

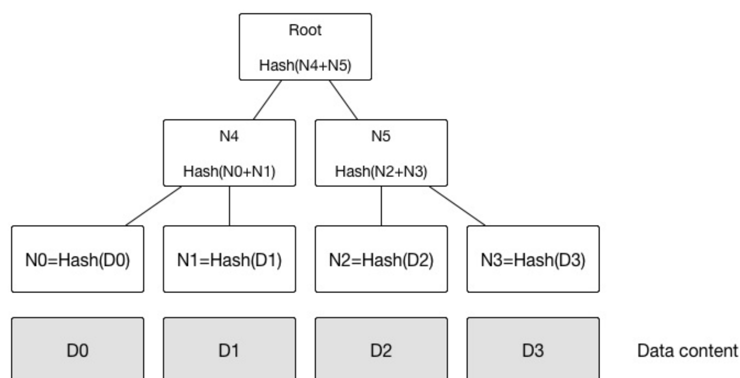


图 *1.8.6.1 - Merkle* 树示例

默克尔树（又叫哈希树）是一种二叉树，由一个根节点、一组中间节点和一组叶节点组成。最下面的叶节点包含存储数据或其哈希值，每个中间节点是它的两个孩子节点内容的哈希值，根节点也是由它的两个子节点内容的哈希值组成。

进一步的，默克尔树可以推广到多叉树的情形。

默克尔树的特点是，底层数据的任何变动，都会传递到其父亲节点，一直到树根。

默克尔树的典型应用场景包括：

- 快速比较大量数据：当两个默克尔树根相同时，则意味着所代表的数据必然相同。
- 快速定位修改：例如上例中，如果 D1 中数据被修改，会影响到 N1，N4 和 Root。因此，沿着 Root --> N4 --> N1，可以快速定位到发生改变的 D1；
- 零知识证明：例如如何证明某个数据（D0……D3）中包括给定内容 D0，很简单，构造一个默克尔树，公布 N0，N1，N4，Root，D0 拥有者可以很容易检测 D0 存在，但不知道其它内容。

同态加密

定义

同态加密（Homomorphic Encryption）是一种特殊的加密方法，允许对密文进行处理得到仍然是加密的结果，即对密文直接进行处理，跟对明文进行处理再加密，得到的结果相同。从代数的角度讲，即同态性。

如果定义一个运算符 ，对加密算法 `E` 和 解密算法 `D`，满足：

则意味着对于该运算满足同态性。

同态性在代数上包括：加法同态、乘法同态、减法同态和除法同态。同时满足加法同态和乘法同态，则意味着是 `代数同态`，即 `全同态`。同时满足四种同态性，则被称为 `算数同态`。

历史

同态加密的问题最早是由 Ron Rivest、Leonard Adleman 和 Michael L. Dertouzos 在 1978 年提出，但 第一个“全同态”的算法 到 2009 年才被克雷格·金特里（Craig Gentry）证明。

仅满足加法同态的算法包括 Paillier 和 Benaloh 算法；仅满足乘法同态的算法包括 RSA 和 ElGamal 算法。

同态加密在云时代的意义十分重大。目前，从安全角度讲，用户还不敢将敏感信息直接放到第三方云上进行处理。如果有了比较实用的同态加密技术，则大家就可以放心的使用各种云服务了。

遗憾的是，目前已知的同态加密技术需要消耗大量的计算时间，还远达不到实用的水平。

函数加密

与同态加密相关的一个问题是函数加密。

同态加密保护的是数据本身，而函数加密顾名思义保护的是处理函数本身，即让第三方看不到处理过程的前提下，对数据进行处理。

该问题已被证明是不存在对多个通用函数的任意多 key 的方案，目前仅能做到对某个特定函数的一个 key 的方案。

其它问题

零知识证明（zero knowledge validation）

证明者在不向验证者提供任何有用的信息的前提下，使验证者相信某个论断是正确的。

例如，A 向 B 证明自己有一个物品，但 B 无法拿到这个物品，无法用 A 的证明去向别人证明自己也拥有这个物品。

小结

比特币 —— 区块链思想诞生的摇篮

之所以看得更远，是因为站在了巨人的肩膀上。

作为区块链思想诞生的源头，比特币项目值得区块链技术爱好者们仔细研究。

比特币网络是首个得到大规模部署的区块链技术应用，并且是首个得到实践检验的数字货币实现，无论在信息技术历史还是在金融学历史上都具有十分重要的意义。比特币项目在诞生和发展过程中，借鉴了来自数字货币、密码学、博弈论、分布式系统、控制论等多个领域的技术成果，可谓博采众家之长于一身。

虽然后来的区块链技术应用已经远超越了数字货币的范畴，但探索比特币项目的发展历程和设计思路，对于深刻理解区块链技术的来龙去脉有着重要的价值。

本章将介绍比特币项目的来源、核心原理设计、相关的工具，以及关键的技术话题。

比特币项目简介



图 *1.9.1.1* - 比特币项目

比特币（BitCoin，BTC）是基于区块链技术的一种数字货币实现；比特币网络是历史上首个经过大规模长时间检验的数字货币系统。

自 2009 年正式上线以来，比特币价格经历了数次的震荡，目前每枚比特币市场价格超过 2500 美金。比特币网络中总区块数接近 48 万个。

比特币网络在功能上具有如下特点：

- 去中心化：意味着没有任何独立个体可以对网络中交易进行破坏，任何交易请求都需要大多数参与者的共识；
- 匿名性：比特币网络中账户地址是匿名的，无法从交易信息关联到具体的个体，但这也意味着很难进行审计；
- 通胀预防：比特币的发行需要通过挖矿计算来进行，发行量每四年减半，总量上限为 2100 万枚，无法被超发。

下图来自 blockchain.info 网站，可以看到比特币字诞生以来的汇率（以美元为单位）变化历史。



图 *1.9.1.2* - 比特币汇率历史

比特币大事记

2008 年 11 月 1 日 19:16:33，中本聪在 metzdowd 的加密技术邮件列表发布比特币白皮书：《Bitcoin: A Peer-to-Peer Electronic Cash System》（《比特币：一种点对点的电子现金系统》）。

2009 年 1 月 3 日 18:15:05，中本聪在位于芬兰赫尔辛基（Helsinki）的一个小型服务器上挖出了第一批 50 个比特币，并记录下当天泰晤士报的头版标题：“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks（财政大臣考虑再次紧急援助银行危机）”。第一个区块被称为创世区块或初始区块（Genesis Block），可以通过 https://blockchain.info/block-index/14849 查看其详细内容。

2010 年 5 月 21 日，第一次比特币交易：佛罗里达程序员 Laszlo Hanyecz 用 1 万 BTC 购买了价值 25 美元的披萨优惠券。这是比特币的首个兑换汇率：1: 0.0025 美金。这些比特币在今日价值超过 8000 万美金。

2010 年 7 月 17 日，第一个比特币交易平台成立。

2011 年 4 月，首个有官方正式记载的版本 0.3.21 发布，支持普通用户参与到 P2P 网络中，并开始支持最小单位“聪”。

2011 年，开始出现基于显卡的挖矿设备。2011 年年底，比特币价格约为 2 美元。

2012 年 6 月，Coinbase 成立，支持比特币相关交易。公司目前已经发展为全球数字资产交易平台，同时支持包括比特币、以太币等数字货币。

2012 年 9 月 27 日，比特币基金创立，此时比特币价格为 12.46 美元。

2012 年 11 月 28 日，比特币产量第一次减半。

2013 年 3 月，1/3 的专业矿工已经采用专用 ASIC 矿机进行挖矿。

2013 年 3 月 12 日，比特币发布 0.8.0 版本，大量完善了节点内部管理和网络通信，使得比特币有能力支持后来大规模的 P2P 网络。该版本包含一个严重的 bug，虽然后来被紧急修复，仍造成比特币价格大幅下跌。

2013年 4 月 10 日，BTC 创下历史新高，266 美元。

2013 年 6 月 27 日，德国会议作出决定：持有比特币一年以上将予以免税，被业内认为此举变相认可了比特币的法律地位，此时比特币价格为 102.24 美元。

2013 年 10 月，世界第一台可以兑换比特币的 ATM 在加拿大上线。

2013 年 11 月 29 日，比特币的交易价格创下 1242 美元的历史新高，而同时黄金价格为一盎司 1241.98 美元，比特币价格首度超过黄金。

2014 年 2 月，全球最大比特币交易平台 Mt.Gox 宣告因 85 万个比特币被盗而破产并关闭，造成大量投资者的损失，比特币价格一度暴跌。

2014 年 3 月，中国第一台可以兑换比特币的 ATM 在香港上线。

2014 年 6 月，美国加州通过 AB-129 法案，允许比特币等数字货币在加州进行流通。

2015 年 6 月，纽约成为美国第一个正式进行数字货币监管的州。

2015 年 10 月，欧盟法院裁定比特币交易免征增值税。

2015 年 10月，《经济学人》杂志发表封面文章《信任机器》，开始关注比特币网络背后的区块链技术。

2016 年 1 月，中国人民银行在京召开了数字货币研讨会，会后发布公告宣称或推出数字货币。

2016 年 7 月 9 日，比特币产量第二次减半。

2016 年 8 月 3 日，知名比特币交易所 Bitfinex 遭遇安全攻击，按照当时市值计算，损失超过 6000 万美金。

2017 年 1 月 24 日，中国三大交易所（Okcoin、火币、BTCC）开始收取比特币交易手续费，为成交金额的 0.2%。

2017 年 7 月，比特币网络全网算力首次突破 6 exahash/s（即每秒10的18次方哈希），创下历史新高。

时至今日，单个比特币价格一度接近 20000 美元，总市值超过 2000 亿美金。

比特币区块链目前生成了约 47 万个区块，完整存储需要约 110 GB 的空间，每天普遍完成 20~30 万笔交易。主流的交易所包括 Bitstamp、BTC-e、Bitfinex 等。多家投资机构（包括红杉、IDG、软银、红点等）都投资了比特币相关的创业团队。

其它数字货币

比特币的“成功”，刺激了相关的生态和社区发展，大量类似数字货币（超过 700 种）纷纷出现，比较出名的包括以太币和瑞波（Ripple）币等。

这些数字货币，要么建立在自己独立的区块链网络上，要么复用已有的区块链（例如比特币网络）系统。全球活跃的数字货币用户据称在 290 万 ~ 580 万之间（参考剑桥大学 Judge 商学院 2017 年 4 月发表的《GLOBAL CRYPTOCURRENCY BENCHMARKING STUDY（全球加密货币基准研究）》报告）。

注：通过 *blockchain.info* 网站可以实时查询到比特币网络的状态信息，包括区块、交易在内的详细数据。

实体货币到加密数字货币

区块链最初的思想，诞生于无数先哲对于用加密数字货币替代实体货币的探讨和设计中。

货币的历史演化

众所周知，货币是人类文明发展过程中的一大发明。其最重要的职能包括价值尺度、流通手段、贮藏手段等。很难想象离开了货币，现代社会庞大而复杂的经济和金融体系如何保持运转。也正是因为它如此重要，货币的设计和发行机制是关系到国计民生的大事。

历史上，在自然和人为因素的干预下，货币的形态经历了多个阶段的演化，包括实物货币、金属货币、代用货币、信用货币、电子货币、数字货币等。近代以前相当长的一段时间里，货币的形态一直是以实体的形式存在，可统称为“实体货币”。计算机诞生后，为货币的虚拟化提供了可能性。

同时，货币自身的价值依托也不断发生演化，从最早的实物价值、发行方信用价值，直到今天的对科学技术和信息系统（包括算法、数学、密码学、软件等）的信任价值。

注：中国最早关于货币的确切记载“夏后以玄币”，出现在恒宽《盐铁论·错币》。

纸币的缺陷

理论上，一般等价物都可以作为货币使用。当今世界最常见的货币制度是纸币本位制，因为纸质货币既方便携带、不易仿制，又相对容易辩伪。

或许有人会认为信用卡等电子方式，相对于纸币等货币形式使用起来更为方便。确实，信用卡在某些场景下会更为便捷，但它依赖背后的集中式支付体系，一旦碰到支付系统故障、断网、缺乏支付终端等情况，信用卡就无法使用；另外，信用卡形式往往还需要额外的终端设备支持。

目前，无论是货币形式，还是信用卡形式，都需要额外的支持机构（例如银行）来完成生产、分发、管理等操作。“中心化”的结构带来了管理和监管上的便利，但系统安全性方面存在很大挑战。诸如伪造、信用卡诈骗、盗刷、转账等安全事件屡见不鲜。

很显然，如果能实现一种数字化的货币，保持既有货币方便易用的特性，同时消除纸质货币的缺陷，无疑将极大提高社会整体经济活动的运作效率。

让我们来对比现有的数字货币（以比特币为例）和现实生活中的纸币，两者的优劣如下表所示。

属性	分析	优势方
便携	大部分场景（特别较大数额支付时）下数字货币将具备更好的便携性。	数字货币
防伪	两者各有千秋，但数字货币整体上会略胜一筹。纸币依靠的是各种设计（纸张、油墨、暗纹、夹层等）上的精巧，数字货币依靠的则是密码学上的保障。事实上，纸币的伪造时有发生，但数字货币的伪造目前还无法实现。	数字货币
辩伪	纸币即使依托验钞机等专用设备仍会有误判情况，数字货币依靠密码学易于校验。数字货币胜出。	数字货币
匿名	通常情况下，两者都能提供很好的匿名性。但都无法防御有意的追踪。	持平
交易	对纸币来说，谁物理持有纸币谁就是合法拥有者，交易通过纸币自身的转移即可完成，无法复制。对数字货币来说则复杂得多，因为任何数字物品都是可以被复制的，但数字形式也意味着转移成本会更低。总体看，两者适用不同的情景。	持平
资源	通常情况下，纸币的生产成本要远低于面额。数字货币消耗资源的计算则复杂的多。以比特币为例，最坏情况下可能需要消耗接近甚至超过其面值的电能。	纸币
发行	纸币的发行需要第三方机构的参与；数字货币则通过分布式算法来完成发行。在人类历史上，通胀和通缩往往是不合理地发行货币造成的；数字货币尚缺乏大规模验证，还有待观察。	持平
管理	纸币发行和回收往往通过统一机构，易于监管和审计；而目前数字货币在这方面还缺乏足够支持和验证。	纸币

可见，数字货币并非在所有领域都优于已有的货币形式。要比较两者的优劣应该针对具体情况具体分析。不带前提地鼓吹数字货币并不是一种科学和严谨的态度。实际上，仔细观察数字货币的应用情况就会发现，虽然以比特币为代表的数字货币已在众多领域得到应用，但目前还没有任何一种数字货币能完全替代已有货币。

另外，虽然当前的数字货币“实验”已经取得了不小影响，但可见的局限也很明显：其依赖的区块链和分布式账本技术还缺乏大规模场景的考验；系统的性能和安全性还有待提升；资源的消耗过高；对监管和审计支持不足等。这些问题的解决，都有待金融科技的进一步发展。

注：严格来讲，货币（*money*）不等于现金或通货（*cash/currency*），货币的含义范围更广。

“去中心化”的技术难关

虽然数字货币带来的预期优势可能很美好，但要设计和实现一套能经得住实用考验的数字货币并非易事。

现实生活中常用的纸币具备良好的可转移性，可以相对容易地完成价值的交割。但是对于数字货币来说，因为数字化内容容易被复制，数字货币持有人可以试图将同一份货币发给多个接收者，这种攻击被称为“双重支付攻击（Double-spend Attack）”。

也许有人会想到，银行中的货币实际上也是数字化的，因为通过电子账号里面的数字记录了客户的资产。说的没错，这种电子货币模式有人称为“数字货币 1.0”，它实际上依赖了一个前提：假定存在一个安全可靠的第三方记账机构负责记账，这个机构负责所有的担保所有的环节，最终完成交易。

中心化控制下，数字货币的实现相对容易。但是，有些时候很难找到一个安全可靠的第三方机构，来充当这个记账者角色。

例如，发生贸易的两国可能缺乏足够的外汇储备用以支付；汇率的变化等导致双方对合同有不同意见；网络上的匿名双方进行直接买卖而不通过电子商务平台；交易的两个机构彼此互不信任，找不到双方都认可的第三方担保；使用第三方担保系统，但某些时候可能无法连接；第三方的系统可能会出现故障或被篡改攻击……

这个时候，就只有实现去中心化（De-centralized）或多中心化（Multi-centralized）的数字货币系统。在“去中心化”的场景下，实现数字货币存在如下几个难题：

- 货币的防伪：谁来负责对货币的真伪进行鉴定；
- 货币的交易：如何确保货币从一方安全转移到另外一方；
- 避免双重支付：如何避免同一份货币支付给多个接收者。

可见，在不存在第三方记账机构的情况下，实现一个数字货币系统的挑战着实不小。能否通过技术创新来解决这个难题呢？

比特币融合了数十年在金融、密码学和分布式系统领域的科技成果，首次实现了可以在全球范围内运行的大规模加密货币系统。

原理和设计

比特币网络是一个分布式的点对点网络，网络中的矿工通过“挖矿”来完成对交易记录的记账过程，维护网络的正常运行。

区块链网络提供一个公共可见的记账本，该记账本并非记录每个账户的余额，而是用来记录发生过的交易的历史信息。该设计可以避免重放攻击，即某个合法交易被多次重新发送造成攻击。

基本交易过程

比特币中没有账户的概念。因此，每次发生交易，用户需要将交易记录写到比特币网络账本中，等网络确认后即可认为交易完成。

除了挖矿获得奖励的 coinbase 交易只有输出，正常情况下每个交易需要包括若干输入和输出，未经使用（引用）的交易的输出（Unspent Transaction Outputs，UTXO）可以被新的交易引用作为其合法的输入。被使用过的交易的输出（Spent Transaction Outputs，STXO），则无法被引用作为合法输入。

因此，比特币网络中一笔合法的交易，必须是引用某些已存在交易的 UTXO（必须是属于付款方才能合法引用）作为新交易的输入，并生成新的 UTXO（将属于收款方）。

那么，在交易过程中，付款方如何证明自己所引用的 UTXO 合法？比特币中通过“签名脚本”来实现，并且指定“输出脚本”来限制将来能使用新 UTXO 者只能为指定收款方。对每笔交易，付款方需要进行签名确认。并且，对每一笔交易来说，总输入不能小于总输出。总输入相比总输出多余的部分称为交易费用（Transaction Fee），为生成包含该交易区块的矿工所获得。目前规定每笔交易的交易费用不能小于 0.0001 BTC，交易费用越高，越多矿工愿意包含该交易，也就越早被放到网络中。交易费用在奖励矿工的同时，也避免了网络受到大量攻击。

交易中金额的最小单位是“聪”，即一亿分之一（10^-8）比特币。

下图展示了一些简单的示例交易。更一般情况下，交易的输入、输出可以为多方。

交易	目的	输入	输出	签名	差额
T0	A 转给 B	他人向 A 交易的输出	B 账户可以使用该交易	A 签名确认	输入减输出，为交易服务费
T1	B 转给 C	T0 的输出	C 账户可以使用该交易	B 签名确认	输入减输出，为交易服务费
...	X 转给 Y	他人向 X 交易的输出	Y 账户可以使用该交易	X 签名确认	输入减输出，为交易服务费

需要注意，刚放进网络中的交易（深度为 0）并非是实时得到确认的。进入网络中的交易存在被推翻的可能性，一般要再生成几个新的区块后（深度大于 0）才认为该交易被确认。

下面分别介绍比特币网络中的重要概念和主要设计思路。

重要概念

账户/地址

比特币采用了非对称的加密算法，用户自己保留私钥，对自己发出的交易进行签名确认，并公开公钥。

比特币的账户地址其实就是用户公钥经过一系列 Hash（HASH160，或先进行 SHA256，然后进行 RIPEMD160）及编码运算后生成的 160 位（20 字节）的字符串。

一般地，也常常对账户地址串进行 Base58Check 编码，并添加前导字节（表明支持哪种脚本）和 4 字节校验字节，以提高可读性和准确性。

注：账户并非直接是公钥内容，而是 *Hash* 后的值，避免公钥过早公开后导致被破解出私钥。

交易

交易是完成比特币功能的核心概念，一条交易可能包括如下信息：

- 付款人地址：合法的地址，公钥经过 SHA256 和 RIPEMD160 两次 Hash，得到 160 位 Hash 串；
- 付款人对交易的签字确认：确保交易内容不被篡改；
- 付款人资金的来源交易 ID：从哪个交易的输出作为本次交易的输入；
- 交易的金额：多少钱，跟输入的差额为交易的服务费；
- 收款人地址：合法的地址；
- 时间戳：交易何时能生效。

网络中节点收到交易信息后，将进行如下检查：

- 交易是否已经处理过；
- 交易是否合法。包括地址是否合法、发起交易者是否是输入地址的合法拥有者、是否是 UTXO；
- 交易的输入之和是否大于输出之和。

检查都通过，则将交易标记为合法的未确认交易，并在网络内进行广播。

用户可以从 blockchain.info 网站查看实时的交易信息，一个示例交易的内容如下图所示。

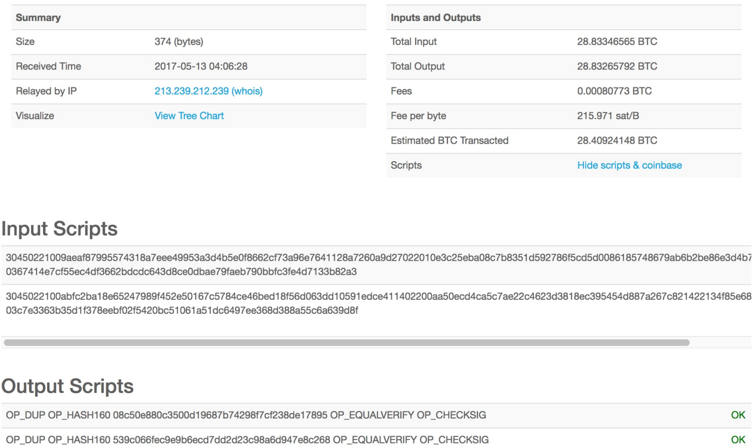


图 *1.9.3.1* - 比特币交易的例子

交易脚本

脚本（Script） 是保障交易完成（主要用于检验交易是否合法）的核心机制，当所依附的交易发生时被触发。通过脚本机制而非写死交易过程，比特币网络实现了一定的可扩展性。比特币脚本语言是一种非图灵完备的语言，类似 Forth 语言。

一般每个交易都会包括两个脚本：负责输入的解锁脚本（scriptSig）和负责输出的锁定脚本（scriptPubKey）。

输出脚本一般由付款方对交易设置锁定，用来对能动用这笔交易的输出（例如，要花费该交易的输出）的对象（收款方）进行权限控制，例如限制必须是某个公钥的拥有者才能花费这笔交易。

认领脚本则用来证明自己可以满足交易输出脚本的锁定条件，即对某个交易的输出（比特币）的拥有权。

输出脚本目前支持两种类型：

- P2PKH：Pay-To-Public-Key-Hash，允许用户将比特币发送到一个或多个典型的比特币地址上（证明拥有该公钥），前导字节一般为 0x00；
- P2SH：Pay-To-Script-Hash，支付者创建一个输出脚本，里边包含另一个脚本（认领脚本）的哈希，一般用于需要多人签名的场景，前导字节一般为 0x05；

以 P2PKH 为例，输出脚本的格式为

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

其中，OP_DUP 是复制栈顶元素；OP_HASH160 是计算 hash 值；OP_EQUALVERIFY 判断栈顶两元素是否相等；OP_CHECKSIG 判断签名是否合法。这条指令实际上保证了只有 pubKey 的拥有者才能合法引用这个输出。

另外一个交易如果要花费这个输出，在引用这个输出的时候，需要提供认领脚本格式为

```
scriptSig: <sig> <pubKey>
```

其中， 是拿 pubKey 对应的私钥对交易（全部交易的输出、输入和脚本）Hash 值进行签名，pubKey 的 Hash 值需要等于 pubKeyHash。

进行交易验证时，会按照先 scriptSig 后 scriptPubKey 的顺序进行依次入栈处理，即完整指令为：

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

读者可以按照栈的过程来进行推算，理解整个脚本的验证过程。

引入脚本机制带来了灵活性，但也引入了更多的安全风险。比特币脚本支持的指令集十分简单，基于栈的处理方式，并且非图灵完备，此外还添加了额外的一些限制（大小限制等）。

区块

比特币区块链的一个区块不能超过 1 MB，将主要包括如下内容：

- 区块大小：4 字节；
- 区块头：80 字节：
- 交易个数计数器：1~9 字节；
- 所有交易的具体内容，可变长，匹配 Merkle 树叶子节点顺序。

其中，区块头信息十分重要，包括：

- 版本号：4 字节；
- 上一个区块头的 Hash 值：链接到上一个合法的块上，对其区块头进行两次 SHA256 操作，32 字节；
- 本区块所包含的所有交易的 Merkle 树根的哈希值：两次 SHA256 操作，32 字节；
- 时间戳：4 字节；
- 难度指标：4 字节；
- Nonce：4 字节，PoW 问题的答案。

可见，要对区块链的完整性进行检查，只需要检验各个区块头部信息即可，无需获取到具体的交易内容，这也是简单交易验证（Simple Payment Verification，SPV）的基本原理。另外，通过头部的链接，提供时序关系的同时加大了对区块中数据进行篡改的难度。

一个示例区块如下图所示。

Block #466150

Summary	
Number Of Transactions	3189
Output Total	4,355.14636223 BTC
Estimated Transaction Volume	581.83177764 BTC
Transaction Fees	1.22474892 BTC
Height	466150 (Main Chain)
Timestamp	2017-05-13 04:35:04
Received Time	2017-05-13 04:35:04
Relayed By	SlushPool
Difficulty	559,970,892,890.84
Bits	402781863
Size	998.171 KB
Version	0x20000002
Nonce	509067181
Block Reward	12.5 BTC

Hashes	
Hash	0000000000000000001cfb77a1a792497664d8ed12058f52c55214abad07e1724
Previous Block	0000000000000000000b43c30612a3368310824e2101e79ad8498a8e6bf41d4fc
Next Block(s)	
Merkle Root	a26cff904e6f73806f2ed18e1abb89c21ba426cfc41495a92313cb4a24b4032f

图 *1.9.3.2* - 比特币区块的例子

创新设计

比特币在设计上提出了很多创新点，主要考虑了避免作恶、采用负反馈调节和基于概率的共识机制等三个方面。

如何避免作恶

基于经济博弈原理。在一个开放的网络中，无法通过技术手段保证每个人都是合作的。但可以通过经济博弈来让合作者得到利益，让非合作者遭受损失和风险。

实际上，博弈论早已被广泛应用到众多领域。

一个经典的例子是两个人来分一个蛋糕，如果都想拿到较大的一块，在没有第三方的前提下，该怎么制定规则才公平？

最简单的一个方案是任意一个人负责分配蛋糕，并且这个人后挑选。

注：如果推广到 *N* 个人呢？

比特币网络中所有试图参与者（矿工）都首先要付出挖矿的代价，进行算力消耗，越想拿到新区块的决定权，意味着抵押的算力越多。一旦失败，这些算力都会被没收掉，成为沉没成本。当网络中存在众多参与者时，个体试图拿到新区块决定权要付出的算力成本是巨大的，意味着进行一次作恶付出的代价已经超过可能带来的好处。

负反馈调节

比特币网络在设计上，很好的体现了负反馈的控制论基本原理。

比特币网络中矿工越多，系统就越稳定，比特币价值就越高，但挖到矿的概率会降低。

反之，网络中矿工减少，会让系统更容易导致被攻击，比特币价值越低，但挖到矿的概率会提高。

因此，比特币的价格理论上应该稳定在一个合适的值（网络稳定性也会稳定在相应的值），这个价格乘以挖到矿的概率，恰好达到矿工的收益预期。

从长远角度看，硬件成本是下降的，但每个区块的比特币奖励每隔 4 年减半，最终将在 2140 年达到 2100 万枚，之后将完全依靠交易的服务费来鼓励矿工对网络的维护。

注：比特币最小单位是“聪”，即 *10^(-8)* 比特币，总“聪”数为 *2.1E15*。对于 *64* 位处理器来说，高精度浮点计数的限制导致单个数值不能超过 *2^53* 约等于 *9E15*。

共识机制

传统共识问题往往是考虑在一个相对封闭的分布式系统中，允许同时存在正常节点、故障节点，如何快速达成一致。

对于比特币网络来说，它是完全开放的，可能面向各种攻击情况，同时基于 Internet 的网络质量只能保证“尽力而为”，导致问题更加复杂，传统的一致性算法在这种场景下难以实用。

因此，比特币网络不得不对共识的目标和过程都进行了一系列限制，提出了基于 Proof of Work（PoW）的共识机制。

首先是不实现面向最终确认的共识，而是基于概率、随时间逐步增强确认的共识。现有达成的结果在理论上都可能被推翻，只是攻击者要付出的代价随时间而指数级上升，被推翻的可能性随之指数级的下降。

此外，考虑到 Internet 的尺度，达成共识的时间相对比较长。按照区块（一组交易）来进行阶段性的确认（快照），提高网络整体的可用性。

最后，限制网络中共识的噪音。通过进行大量的 Hash 计算和少数的合法结果来限制合法提案的个数，进一步提高网络中共识的稳定性。

挖矿

基本原理

了解比特币，最应该知道的一个概念就是“挖矿”。挖矿是参与维护比特币网络的节点，通过协助生成新区块来获取一定量新增的比特币的过程。

当用户向比特币网络中发布交易后，需要有人将交易进行确认，形成新的区块，串联到区块链中。在一个互相不信任的分布式系统中，该由谁来完成这件事情呢？比特币网络采用了“挖矿”的方式来解决这个问题。

目前，每 10 分钟左右生成一个不超过 1 MB 大小的区块（记录了这 10 分钟内发生的验证过的交易内容），串联到最长的链尾部，每个区块的成功提交者可以得到系统 12.5 个比特币的奖励（该奖励作为区块内的第一个交易，一定区块数后才能使用），以及用户附加到交易上的支付服务费用。即便没有任何用户交易，矿工也可以自行产生合法的区块并获得奖励。

每个区块的奖励最初是 50 个比特币，每隔 21 万个区块自动减半，即 4 年时间，最终比特币总量稳定在 2100 万个。因此，比特币是一种通缩的货币。

挖矿过程

挖矿的具体过程为：参与者综合上一个区块的 Hash 值，上一个区块生成之后的新的验证过的交易内容，再加上自己猜测的一个随机数 X，一起打包到一个候选新区块，让新区块的 Hash 值小于比特币网络中给定的一个数。这是一道面向全体矿工的“计算题”，这个数越小，计算出来就越难。

系统每隔两周（即经过 2016 个区块）会根据上一周期的挖矿时间来调整挖矿难度（通过调整限制数的大小），来调节生成区块的时间稳定在 10 分钟左右。为了避免震荡，每次调整的最大幅度为 4 倍。历史上最快的出块时间小于 10s，最慢的出块时间超过 1 个小时。

为了挖到矿，参与处理区块的用户端往往需要付出大量的时间和计算力。算力一般以每秒进行多少次 Hash 计算为单位，记为 h/s。目前，比特币网络算力峰值已经达到了每秒数百亿亿次。

汇丰银行分析师 Anton Tonev 和 Davy Jose 曾表示，比特币区块链（通过挖矿）提供了一个局部的、迄今为止最优的解决方案：如何在分散的系统中验证信任。这就意味着，区块链本质上解决了传统依赖于第三方的问题，因为这个协议不只满足了中心化机构追踪交易的需求，还使得陌生人之间产生信任。区块链的技术和安全的过程使得陌生人之间在没有被信任的第三方时产生信任。

如何看待挖矿

2010 年以前，挖矿还是一个非常热门的盈利行业。

但是随着相关技术和设备的发展，现在个人进行挖矿的收益已经降得很低。从概率上说，由于当前参与挖矿的计算力实在过于庞大（已经超出了大部分的超算中心），一般的算力已经不可能挖到比特币。特别那些想着利用虚拟机来挖矿的想法，意义确实不大了。

从普通的 CPU（2009 年）、到后来的 GPU（2010 年） 和 FPGA（2011 年末）、到后来的 ASIC 矿机（2013 年年初，目前单片算力已达每秒数百亿次 Hash 计算）、再到现在众多矿机联合组成矿池（知名矿池包括 F2Pool、BitFury、BTCC 等）。短短数年间，比特币矿机的技术走完了过去几十年的集成电路技术进化历程，并且还颇有创新之处。确实是哪里有利益，哪里的技术就飞速发展！目前，矿机主要集中在中国大陆（超过一半的算力）和欧美，大家比拼的是一定计算性能情况下低电压和低功耗的电路设计。全网的算力已超过每秒 10^18 次 Hash 计算。

很自然地，读者可能会想到，如果有人掌握了强大的计算力，计算出所有的新区块，并且拒不承认他人的交易内容，那是不是就能破坏掉比特币网络。确实如此，基本上个体达到 1/3 的计算力，比特币网络就存在被破坏的风险了；达到 1/2 的算力，从概率上就掌控整个网络了。但是要实现这么大的算力，将需要付出巨大的经济成本。

那么有没有办法防护呢？除了尽量避免计算力放到同一个组织手里，没太好的办法，这是目前 PoW 机制自身造成的。

也有人认为为了共识区块的生成，大部分计算力（特别是最终未能算出区块的算力）其实都浪费了。有人提出用 PoS（Proof of Stake）和 DPoS 等协议，利用权益证明（例如持有货币的币龄）作为衡量指标进行投票，相对 PoW 可以节约大量的能耗。PoS 可能会带来囤积货币的问题。除此之外，还有活跃度证明（Proof of Activity，PoA）、消耗证明（Proof of Burn，PoB）、能力证明（Proof of Capacity, PoC）、消逝时间证明（Proof of Elapsed Time）、股权速率证明（Proof of Stake Velocity，PoSV）等，采用了不同的衡量指标。

当然，无论哪种机制，都无法解决所有问题。一种可能的优化思路是引入随机代理人制度，通过算法在某段时间内确保只让部分节点参加共识的提案，并且要发放一部分“奖励”给所有在线贡献的节点。

共识机制

比特币网络是完全公开的，任何人都可以匿名接入，因此共识协议的稳定性和防攻击性十分关键。

比特币区块链采用了 Proof of Work（PoW）的机制来实现共识，该机制最早于 1998 年在 B-money 设计中提出。

目前，Proof of X 系列中比较出名的一致性协议包括 PoW、PoS 和 DPoS 等，都是通过经济惩罚来限制恶意参与。

工作量证明

工作量证明，通过计算来猜测一个数值（nonce），使得拼凑上交易数据后内容的 Hash 值满足规定的上限（来源于 hashcash）。由于 Hash 难题在目前计算模型下需要大量的计算，这就保证在一段时间内，系统中只能出现少数合法提案。反过来，能够提出合法提案，也证明提案者确实已经付出了一定的工作量。

同时，这些少量的合法提案会在网络中进行广播，收到的用户进行验证后，会基于用户认为的最长链基础上继续难题的计算。因此，系统中可能出现链的分叉（Fork），但最终会有一条链成为最长的链。

Hash 问题具有不可逆的特点，因此，目前除了暴力计算外，还没有有效的算法进行解决。反之，如果获得符合要求的 nonce，则说明在概率上是付出了对应的算力。谁的算力多，谁最先解决问题的概率就越大。当掌握超过全网一半算力时，从概率上就能控制网络中链的走向。这也是所谓 `51%` 攻击的由来。

参与 PoW 计算比赛的人，将付出不小的经济成本（硬件、电力、维护等）。当没有最终成为首个算出合法 nonce 值的“幸运儿”时，这些成本都将被沉没掉。这也保障了，如果有人尝试恶意破坏，需要付出大量的经济成本。也有设计试图将后算出结果者的算力按照一定比例折合进下一轮比赛考虑。

有一个很直观的超市付款的例子，可以说明为何这种经济博弈模式会确保系统中最长链的唯一性。

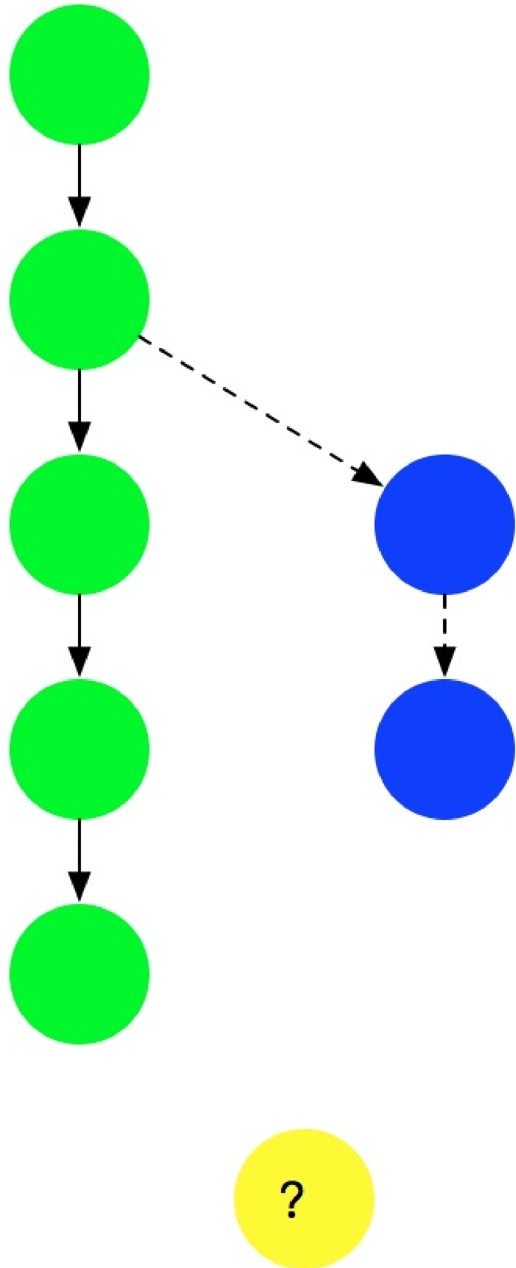


图 *1.9.5.1 - Pow* 保证一致性

假定超市只有一个出口，付款时需要排成一队，可能有人不守规矩要插队。超市管理员会检查队伍，认为最长的一条队伍是合法的，并让不合法的分叉队伍重新排队。新到来的人只要足够理智，就会自觉选择最长的队伍进行排队。这是因为，看到多条链的参与者往往认为目前越长的链具备越大的胜出可能性，从而更倾向于选择长的链。

权益证明

权益证明（Proof of Stake，PoS），最早在 2013 年被提出，最早在 Peercoin 系统中被实现，类似现实生活中的股东机制，拥有股份越多的人越容易获取记账权（同时越倾向于维护网络的正常工作）。

典型的过程是通过保证金（代币、资产、名声等具备价值属性的物品即可）来对赌一个合法的块成为新的区块，收益为抵押资本的利息和交易服务费。提供证明的保证金（例如通过转账货币记录）越多，则获得记账权的概率就越大。合法记账者可以获得收益。

PoS 试图解决在 PoW 中大量资源被浪费的缺点，受到了广泛关注。恶意参与者将存在保证金被罚没的风险，即损失经济利益。

一般的，对于 PoS 来说，需要掌握超过全网 1/3 的资源，才有可能左右最终的结果。这个也很容易理解，三个人投票，前两人分别支持一方，这时候，第三方的投票将决定最终结果。

PoS 也有一些改进的算法，包括授权股权证明机制（DPoS），即股东们投票选出一个董事会，董事会中成员才有权进行代理记账。这些算法在实践中得到了不错的验证，但是并没有理论上的证明。

2017 年 8 月，来自爱丁堡大学和康涅狄格大学的 Aggelos Kiayias 等学者在论文《Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol》中提出了 Ouroboros 区块链共识协议，该协议可以达到诚实行为的近似纳什均衡，认为是首个可证实安全的 PoS 协议。

闪电网络

比特币的交易网络最为人诟病的一点便是交易性能：全网每秒 7 笔左右的交易速度，远低于传统的金融交易系统；同时，等待 6 个块的可信确认将导致约 1 个小时的最终确认时间。

为了提升性能，社区提出了闪电网络等创新的设计。

闪电网络的主要思路十分简单——将大量交易放到比特币区块链之外进行，只把关键环节放到链上进行确认。该设计最早于 2015 年 2 月在论文《The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments》中提出。

比特币的区块链机制自身已经提供了很好的可信保障，但是相对较慢；另一方面考虑，对于大量的小额交易来说，是否真需要这么高的可信性？

闪电网络主要通过引入智能合约的思想来完善链下的交易渠道。核心的概念主要有两个：RSMC（Recoverable Sequence Maturity Contract）和 HTLC（Hashed Timelock Contract）。前者解决了链下交易的确认问题，后者解决了支付通道的问题。

RSMC

Recoverable Sequence Maturity Contract，即“可撤销的顺序成熟度合同”。这个词很绕，其实主要原理很简单，类似资金池机制。

首先假定交易双方之间存在一个“微支付通道”（资金池）。交易双方先预存一部分资金到“微支付通道”里，初始情况下双方的分配方案等于预存的金额。每次发生交易，需要对交易后产生资金分配结果共同进行确认，同时签字把旧版本的分配方案作废掉。任何一方需要提现时，可以将他手里双方签署过的交易结果写到区块链网络中，从而被确认。从这个过程中可以可以看到，只有在提现时候才需要通过区块链。

任何一个版本的方案都需要经过双方的签名认证才合法。任何一方在任何时候都可以提出提现，提现时需要提供一个双方都签名过的资金分配方案（意味着肯定是某次交易后的结果，被双方确认过，但未必是最新的结果）。在一定时间内，如果另外一方拿出证明表明这个方案其实之前被作废了（非最新的交易结果），则资金罚没给质疑方；否则按照提出方的结果进行分配。罚没机制可以确保了没人会故意拿一个旧的交易结果来提现。

另外，即使双方都确认了某次提现，首先提出提现一方的资金到账时间要晚于对方，这就鼓励大家尽量都在链外完成交易。通过 RSMC，可以实现大量中间交易发生在链外。

HTLC

微支付通道是通过 Hashed Timelock Contract 来实现的，中文意思是“哈希的带时钟的合约”。这个其实就是限时转账。理解起来也很简单，通过智能合约，双方约定转账方先冻结一笔钱，并提供一个哈希值，如果在一定时间内有人能提出一个字符串，使得它哈希后的值跟已知值匹配（实际上意味着转账方授权了接收方来提现），则这笔钱转给接收方。

不太恰当的例子，约定一定时间内，有人知道了某个暗语（可以生成匹配的哈希值），就可以拿到这个指定的资金。

推广一步，甲想转账给丙，丙先发给甲一个哈希值。甲可以先跟乙签订一个合同，如果你在一定时间内能告诉我一个暗语，我就给你多少钱。乙于是跑去跟丙签订一个合同，如果你告诉我那个暗语，我就给你多少钱。丙于是告诉乙暗语，拿到乙的钱，乙又从甲拿到钱。最终达到结果是甲转账给丙。这样甲和丙之间似乎构成了一条完整的虚拟的“支付通道”。

HTLC 机制可以扩展到多个人的场景。

闪电网络

RSMC 保障了两个人之间的直接交易可以在链下完成，HTLC 保障了任意两个人之间的转账都可以通过一条“支付”通道来完成。闪电网络整合这两种机制，就可以实现任意两个人之间的交易都在链下完成了。

在整个交易中，智能合约起到了中介的重要角色，而区块链网络则确保最终的交易结果被确认。

侧链

侧链（Sidechain）协议允许资产在比特币区块链和其他区块链之间互转。这一项目也来自比特币社区，最早是在 2013 年 12 月提出，2014 年 4 月立项，由 Blockstream 公司（由比特币核心开发者 Adam Back、Matt Corallo 等共同发起成立）主导研发。侧链协议于 2014 年 10 月在白皮书《Enabling Blockchain Innovations with Pegged Sidechains》中公开。

侧链诞生前，众多“山寨币”的出现正在碎片化整个数字货币市场，再加上以太坊等项目的竞争，一些比特币开发者希望能借助侧链的形式扩展比特币的底层协议。

简单来讲，以比特币区块链作为主链（Parent chain），其他区块链作为侧链，二者通过双向挂钩（Two-way peg），可实现比特币从主链转移到侧链进行流通。

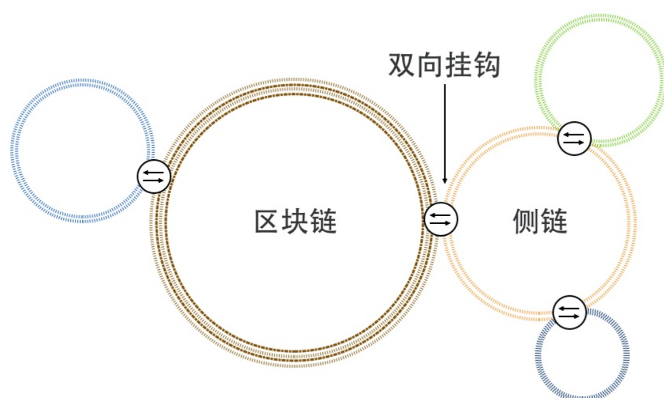


图 *1.9.7.1* - 比特币侧链

侧链可以是一个独立的区块链，有自己按需定制的账本、共识机制、交易类型、脚本和合约的支持等。侧链不能发行比特币，但可以通过支持与比特币区块链挂钩来引入和流通一定数量的比特币。当比特币在侧链流通时，主链上对应的比特币会被锁定，直到比特币从侧链回到主链。可以看到，侧链机制可将一些定制化或高频的交易放到比特币主链之外进行，实现了比特币区块链的扩展。侧链的核心原理在于能够冻结一条链上的资产，然后在另一条链上产生，可以通过多种方式来实现。这里讲解 Blockstream 提出的基于简单支付验证（Simplified Payment Verification，SPV）证明的方法。

SPV 证明

如前面章节所述，在比特币系统中验证交易时，涉及到交易合法性检查、双重花费检查、脚本检查等。由于验证过程需要完整的 UTXO 记录，通常要由运行着完整功能节点的矿工来完成。

而很多时候，用户只关心与自己相关的那些交易，比如当用户收到其他人号称发来的比特币时，只希望能够知道交易是否合法、是否已在区块链中存在了足够的时间（即获得足够的确认），而不需要自己成为完整节点做出完整验证。

中本聪设计的简单支付验证（Simplified Payment Verification，SPV）可以实现这一点。SPV 能够以较小的代价判断某个支付交易是否已经被验证过（存在于区块链中），以及得到了多少算力保护（定位包含该交易的区块在区块链中的位置）。SPV 客户端只需要下载所有区块的区块头（Block Header），并进行简单的定位和计算工作就可以给出验证结论。

侧链协议中，用 SPV 来证明一个交易确实已经在区块链中发生过，称为 SPV 证明（SPV Proof）。一个 SPV 证明包括两部分内容：一组区块头的列表，表示工作量证明；一个特定输出（output）确实存在于某个区块中的密码学证明。

双向挂钩

侧链协议的设计难点在于如何让资产在主链和侧链之间安全流转。简而言之，接受资产的链必须确保发送资产的链上的币被可靠锁定。

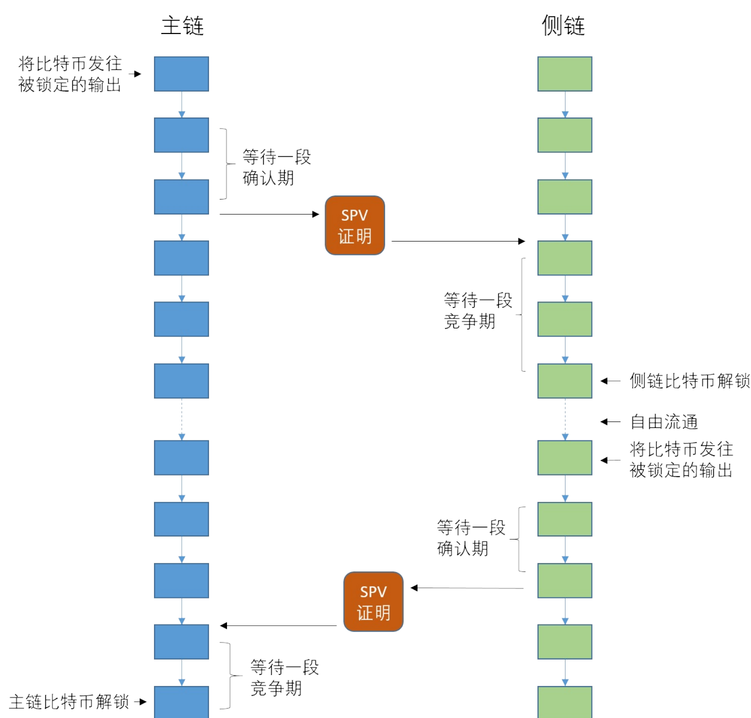


图 *1.9.7.2* - 侧链双向挂钩的过程

具体，协议采用双向挂钩机制实现比特币向侧链转移和返回。主链和侧链需要对对方的特定交易做 SPV 验证。完整过程如下：

- 当用户要向侧链转移比特币时，首先在主链创建交易，待转移的比特币被发往一个特殊的输出。这些比特币在主链上被锁定。

* 等待一段确认期，使得上述交易获得足够的工作量确认。
* 用户在侧链创建交易提取比特币，需要在这笔交易的输入指明上述主链被锁定的输出，并提供足够的 SPV 证明。
* 等待一段竞争期，防止双重花费攻击。
* 比特币在侧链上自由流通。
* 当用户想让比特币返回主链时，采取类似的反向操作。首先在侧链创建交易，待返回的比特币被发往一个特殊的输出。先等待一段确认期后，在主链用足够的对侧链输出的 SPV 证明来解锁最早被锁定的输出。竞争期过后，主链比特币恢复流通。

最新进展

侧链技术最早由 Blockstream 公司进行探索，于 2015 年 10 月联合合作伙伴发布了基于侧链的商业化应用 Liquid。

基于一年多的探索，Blockstream 于 2017 年 1 月发表文章《Strong Federations: An Interoperable Blockchain Solution to Centralized Third Party Risks》，被称为对侧链早期白皮书的补充和改良。白皮书中着重描述了联合挂钩（Federated Pegs）的相关概念和应用。

此外，还有一些其他公司或组织也在探索如何合理地应用侧链技术，包括 ConsenSys、Rootstock、Lisk 等。

热点问题

设计中的权衡

比特币的设计目标在于支持一套安全、开放、分布式的数字货币系统。围绕这一目标，比特币协议的设计中很多地方都体现了权衡（trade-off）的思想。

- 区块容量：更大的区块容量可以带来更高的交易吞吐率，但会增加挖矿成本，带来中心化的风险，同时增大存储的代价。兼顾多方面的考虑，当前的区块容量上限设定为 1MB。
- 出块间隔时间：更短的出块间隔可以缩短交易确认的时间，但也可能导致分叉增多，降低网络可用性。
- 脚本支持程度：更强大的脚本指令集可以带来更多灵活性，但也会引入更多安全风险。

分叉

比特币协议不会一成不变。当需要修复漏洞、扩展功能或调整结构时，比特币需要在全网的配合下进行升级。升级通常涉及更改交易的数据结构或区块的数据结构。

由于分布在全球的节点不可能同时完成升级来遵循新的协议，因此比特币区块链在升级时可能发生分叉（Fork）。对于一次升级，如果把网络中升级了的节点称为新节点，未升级的节点称为旧节点，根据新旧节点相互兼容性上的区别，可分为软分叉（Soft Fork）和硬分叉（Hard Fork）。

- 如果旧节点仍然能够验证接受新节点产生的交易和区块，则称为软分叉。旧节点可能不理解新节点产生的一部分数据，但不会拒绝。网络既向后和向前兼容，因此这类升级可以平稳进行。
- 如果旧节点不接受新节点产生的交易和区块，则称为硬分叉。网络只向后兼容，不向前兼容。这类升级往往引起一段时间内新旧节点所认可的区块不同，分出两条链，直到旧节点升级完成。

尽管通过硬分叉升级区块链协议的难度大于软分叉，但软分叉能做的事情毕竟有限，一些大胆的改动只能通过硬分叉完成。

交易延展性

交易延展性（Transaction Malleablility）是比特币的一个设计缺陷。简单来讲，是指当交易发起者对交易签名（sign）之后，交易 ID 仍然可能被改变。

下面是一个比特币交易的例子。

```
{
  "txid": "f200c37aa171e9687452a2c78f2537f134c307087001745edacb58304053db20",
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "21f10dbfb0ff49e2853629517fa176dc00d943f203aae3511288a7dd89280ac2",
      "vout": 0,
      "scriptSig": {
        "asm": "304402204f7fb0b1e0d154db27dbdeeeb8db7b7d3b887a33e712870503438d8be2d66a
0102204782a2714215dc0d581e1d435b41bc6eced2c213c9ba0f993e7fcf468bb5d311[ALL] 025840d511
c4bc6690916270a54a6e9290fab687f512c18eb2df0428fa69a26299",
        "hex": "47304402204f7fb0b1e0d154db27dbdeeeb8db7b7d3b887a33e712870503438d8be2d6
6a0102204782a2714215dc0d581e1d435b41bc6eced2c213c9ba0f993e7fcf468bb5d3110121025840d511
c4bc6690916270a54a6e9290fab687f512c18eb2df0428fa69a26299"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00167995,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 7c4338dea7964947b3f0954f61ef40502fe8f791 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a9147c4338dea7964947b3f0954f61ef40502fe8f79188ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1CL3KTtkN8KgHAeWMMWfG9CPL3o5FSMU4P"
        ]
      }
    }
  ]
}
```

发起者对交易的签名（scriptSig）位于交易的输入（vin）当中，属于交易内容的一部分。交易 ID（txid）是整个交易内容的 Hash 值。这就造成了一个问题：攻击者（尤其是签名方）可以通过改变 scriptSig 来改变 txid，而交易仍旧保持合法。例如，反转 ECDSA 签名过程中的 S 值，签名仍然合法，交易仍然能够被传播。

这种延展性攻击能改变交易 ID，但交易的输入和输出不会被改变，所以攻击者不会直接盗取比特币。这也是为什么这一问题能在比特币网络中存在如此之久，而仍未被根治。

然而，延展性攻击仍然会带来一些问题。比如，在原始交易未被确认之前广播 ID 改变了的交易可能误导相关方对交易状态的判断，甚至发动拒绝服务攻击；多重签名场景下，一个签名者有能力改变交易 ID，给其他签名者的资产带来潜在风险。同时，延展性问题也会阻碍闪电网络等比特币扩展方案的实施。

扩容之争

比特币当前将区块容量限制在 1MB 以下。如图所示，随着用户和交易量的增加，这一限制已逐渐不能满足比特币的交易需求，使得交易日益拥堵、交易手续费不断上涨。

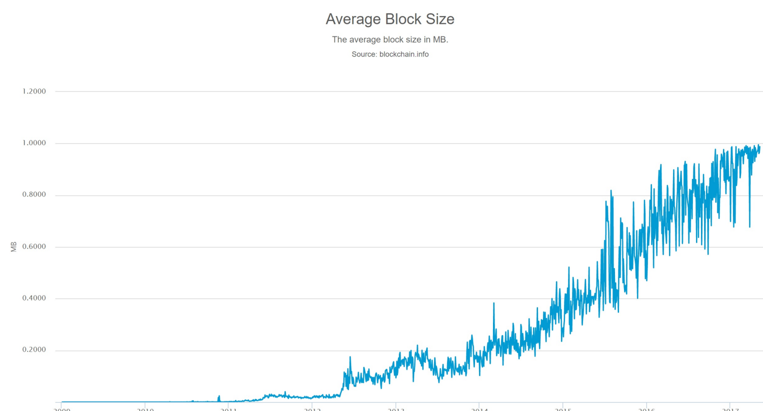


图 *1.9.8.1* - 日益增加的区块容量

关于比特币扩容的持续争论从 2015 年便已开始，期间有一系列方案被摆上台面，包括各种链上扩容提议、用侧链或闪电网络扩展比特币等。考虑到比特币复杂的社区环境，其扩容方案早已不是一方能说了算；而任何一个方案想让要达成广泛共识都比较困难，不同的方案之间也很难调和。

当前，扩容之争主要集中在两派：代表核心开发者的 Bitcoin Core 团队主推的隔离见证方案，和 Bitcoin Unlimited 团队推出的方案。

隔离见证方案

隔离见证（Segregated Witness，简称 SegWit）是指将交易中的签名部分从交易的输入中隔离出来，放到交易末尾的被称为见证（Witness）的字段当中。

对交易 ID 的计算将不再包含这一签名部分，所以这也是延展性问题的一种解法，给引入闪电网络等第二层协议增强了安全性。

同时，隔离见证会将区块容量上限理论上提高到 4MB。对隔离见证的描述可详见五个比特币改进协议（Bitcoin Improvement Proposal）：BIP 141 ~ BIP 145。

Bitcoin Unlimited 方案

Bitcoin Unlimited 方案（简称 BU）是指扩展比特币客户端，使矿工可以自由配置他们想要生成和验证的区块的容量。

根据方案的设想，区块容量的上限会根据众多节点和矿工的配置进行自然收敛。Bitcoin Unlimited Improvement Proposal（BUIP） 001 中表述了这一对比特币客户端的拓展提议，该方案已获得一些大型矿池的支持和部署。

比特币的监管和追踪

比特币的匿名特性，使得其上交易的监管变得十分困难。

不少非法分子利用这一点，通过比特币转移资金。例如 WannaCry 网络病毒向受害者勒索比特币，短短三天时间里传播并影响到全球 150 多个国家。尽管这些不恰当的行为与比特币项目自身并无直接关系，但都或多或少给比特币社区带来了负面影响。

实际上，认为通过比特币就可以实现完全匿名化并不现实。虽然交易账户自身是匿名的 Hash 地址，但一些研究成果（如《An analysis of anonymity in the bitcoin system》）表明，通过分析大量公开可得的交易记录，有很大概率可以追踪到比特币的实际转移路线，甚至可以追踪到真实用户。

相关工具

比特币相关工具包括客户端、钱包和矿机等。

客户端

比特币客户端用于和比特币网络进行交互，同时可以参与到网络的维护。

客户端分为三种：完整客户端、轻量级客户端和在线客户端。

- 完整客户端：存储所有的交易历史记录，功能完备；
- 轻量级客户端：不保存交易副本，交易需要向别人查询；
- 在线客户端：通过网页模式来浏览第三方服务器提供的服务。

比特币客户端可以从 https://bitcoin.org/en/download 下载到。

基于比特币客户端，可以很容易实现用户钱包功能。

钱包

比特币钱包存储和保护用户的私钥，并提供查询比特币余额、收发比特币等功能。根据私钥存储方式不同，钱包主要分为以下几种：

- 离线钱包：离线存储私钥，也称为“冷钱包”。安全性相对最强，但无法直接发送交易，便利性差。
- 本地钱包：用本地设备存储私钥。可直接向比特币网络发送交易，易用性强，但本地设备存在被攻击风险。
- 在线钱包：用钱包服务器存储经用户口令加密过的私钥。易用性强，但钱包服务器同样可能被攻击。
- 多重签名钱包：由多方共同管理一个钱包地址，比如 2 of 3 模式下，集合三位管理者中的两位的私钥便可以发送交易。

比特币钱包可以从 https://bitcoin.org/en/choose-your-wallet 获取到。

矿机

比特币矿机是专门为“挖矿”设计的硬件设备，目前主要包括基于 GPU 和 ASIC 芯片的专用矿机。这些矿机往往采用特殊的设计来加速挖矿过程中的计算处理。

矿机最重要的属性是可提供的算力（通常以每秒可进行 Hash 计算的次数来表示）和所需要的功耗。当算力足够大，可以在概率意义上挖到足够多的新的区块，来弥补电力费用时，该矿机是可以盈利的；当单位电力产生的算力不足以支付电力费用时，该矿机无法盈利，意味着

只能被淘汰。

目前，比特币网络中的全网算力仍然在快速增长中，矿工需要综合考虑算力变化、比特币价格、功耗带来的电费等许多问题，需要算好“经济账”。

本章小结

本章介绍了比特币项目的相关知识，包括核心技术、工具、设计，以及最新的闪电网络、侧链和扩容讨论等进展。

比特币自身作为数字货币领域的重大突破，对分布式记账领域有着很深远的影响。尤其是其底层的区块链技术，已经受到金融和信息行业的重视，在许多场景下都得到应用。

通过本章的剖析，可以看出，比特币网络系统中并没有完全从头进行创新的技术，而是有机地组合了密码学、博弈论、记账技术、分布式系统和网络、控制论等领域的已有成果。有人认为，比特币发明人对于对这些技术的应用也并没有达到十分专业的地步。但正是如此巧妙的组合，让比特币项目能完成这样一件了不起的创举，也体现出了发明者堪比大师的境界。

以太坊 —— 挣脱数字货币的枷锁

君子和而不同。

在区块链领域，以太坊项目同样是十分出名的开源项目。作为公有区块链平台，以太坊将比特币针对数字货币交易的功能进一步进行拓展，面向更为复杂和灵活的应用场景，支持了智能合约（Smart Contract）这一重要特性。

从此，区块链技术的应用场景，从单一基于 UTXO 的数字货币交易，延伸到图灵完备的通用计算领域。用户不再受限于仅能使用比特币脚本所支持的简单逻辑，而是可以自行设计任意复杂的合约逻辑。这就为构建各种多样化的上层应用开启了大门，可谓意义重大。

本章将参照比特币项目来介绍以太坊项目的核心概念和改进设计，以及如何安装客户端和使用智能合约等内容。

以太坊项目简介



图 *1.10.1.1* - 以太坊项目

以太坊（Ethereum）项目的最初目标，是打造一个运行智能合约的平台（Platform for Smart Contract）。该平台支持图灵完备的应用，按照智能合约的约定逻辑自动执行，理想情况下将不存在故障停机、审查、欺诈，以及第三方干预等问题。

以太坊平台目前支持 Golang、C++、Python 等多种语言实现的客户端。由于核心实现上基于比特币网络的核心思想进行了拓展，因此在很多设计特性上都与比特币网络十分类似。

基于以太坊项目，以太坊团队目前运营了一条公开的区块链平台——以太坊网络。智能合约开发者使用官方提供的工具和以太坊专用应用开发语言 Solidity，可以很容易开发出运行在以太坊网络上的“去中心化”应用（Decentralized Application，DApp）。这些应用将运行在以太坊的虚拟机（Ethereum Virtual Machine，EVM）里。用户通过以太币（Ether）来购买燃料（Gas），维持所部署应用的运行。

以太坊项目的官网网站为 ethereum.org，代码托管在 github.com/ethereum。

以太坊项目简史

相对比特币网络自 2009 年上线的历史，以太坊项目要年轻的多。

2013 年底，比特币开发团队中有一些开发者开始探讨将比特币网络中的核心技术，主要是区块链技术，拓展到更多应用场景的可能性。以太坊的早期发明者 Vitalik Buterin 提出应该能运行任意形式（图灵完备）的应用程序，而不仅仅是比特币中受限制的简单脚本。该设计思想并未得到比特币社区的支持，后来作为以太坊白皮书发布。

2014 年 2 月，更多开发者（包括 Gavin Wood、Jeffrey Wilcke 等）加入以太坊项目，并计划在社区开始以众筹形式募集资金，以开发一个运行智能合约的信任平台。

2014 年 7 月，以太币预售，经过 42 天，总共筹集到价值超过 1800 万美金的比特币。随后在瑞士成立以太坊基金会，负责对募集到的资金进行管理和运营；并组建研发团队以开源社区形式进行平台开发。

2015 年 7 月底，以太坊第一阶段 Frontier 正式发布，标志着以太坊区块链网络的正式上线。这一阶段采用类似比特币网络的 PoW 共识机制，参与节点以矿工挖矿形式维护网络；支持上传智能合约。Frontier 版本实现了计划的基本功能，在运行中测试出了一些安全上的漏洞。这一阶段使用者以开发者居多。

2016 年 3 月，第二阶段 Homestead 开始运行（区块数 1150000），主要改善了安全性，同时开始提供图形界面的客户端，提升了易用性，更多用户加入进来。

2016 年 6 月，DAO 基于以太坊平台进行众筹，受到漏洞攻击，造成价值超过 5000 万美金的以太币被冻结。社区最后通过硬分叉（Hard Fork）进行解决。

2017 年 3 月，以太坊成立以太坊企业级联盟（Enterprise Ethereum Alliance，EEA），联盟成员主要来自摩根大通，微软，芝加哥大学和部分创业企业等。

2017 年 11 月，再次暴露多签名钱包漏洞，造成价值 2.8 亿美元的以太币被冻结。

目前，以太坊网络支持了接近比特币网络的交易量，成为广受关注的公有链项目。

后续按照计划将发布第三阶段 Metropolis 和第四阶段 Serenity，主要特性包括支持 PoS 股权证明的共识机制，以降低原先 PoW 机制造成的能耗浪费；以及图形界面的钱包，以提升易用性。

包括 DAO 在内，以太坊网络已经经历了数次大的硬分叉，注意每次硬分叉后的版本对之前版本并不兼容。

主要特点

以太坊区块链底层也是一个类似比特币网络的 P2P 网络平台，智能合约运行在网络中的以太坊虚拟机里。网络自身是公开可接入的，任何人都可以接入并参与网络中数据的维护，提供运行以太坊虚拟机的资源。

跟比特币项目相比，以太坊区块链的技术特点主要包括：

- 支持图灵完备的智能合约，设计了编程语言 Solidity 和虚拟机 EVM；
- 选用了内存需求较高的哈希函数，避免出现强算力矿机、矿池攻击；
- 叔块（Uncle Block）激励机制，降低矿池的优势，并减少区块产生间隔（10 分钟降低到 15 秒左右）；
- 采用账户系统和世界状态，而不是 UTXO，容易支持更复杂的逻辑；
- 通过 Gas 限制代码执行指令数，避免循环执行攻击；
- 支持 PoW 共识算法，并计划支持效率更高的 PoS 算法。

此外，开发团队还计划通过分片（Sharding）方式来解决网络可扩展性问题。

这些技术特点，解决了比特币网络在运行中被人诟病的一些问题，让以太坊网络具备了更大的应用潜力。

核心概念

基于比特币网络的核心思想，以太坊项目提出了许多创新的技术概念，包括智能合约、基于账户的交易、以太币和燃料等。

智能合约

智能合约（Smart Contract）是以太坊中最为重要的一个概念，即以计算机程序的方式来缔结和运行各种合约。最早在上世纪 90 年代，Nick Szabo 等人就提出过类似的概念，但一直依赖因为缺乏可靠执行智能合约的环境，而被作为一种理论设计。区块链技术的出现，恰好补充了这一缺陷。

以太坊支持通过图灵完备的高级语言（包括 Solidity、Serpent、Viper）等来开发智能合约。智能合约作为运行在以太坊虚拟机（Ethereum Virual Machine，EVM）中的应用，可以接受来自外部的交易请求和事件，通过触发运行提前编写好的代码逻辑，进一步生成新的交易和事件，可以进一步调用其它智能合约。

智能合约的执行结果可能对以太坊网络上的账本状态进行更新。这些修改由于经过了以太坊网络中的共识，一旦确认后无法被伪造和篡改。

账户

在之前章节中，笔者介绍过比特币在设计中并没有账户（Account）的概念，而是采用了 UTXO 模型记录整个系统的状态。任何人都可以通过交易历史来推算出用户的余额信息。而以太坊则采用了不同的做法，直接用账户来记录系统状态。每个账户存储余额信息、智能合约代码和内部数据存储等。以太坊支持在不同的账户之间转移数据，以实现更为复杂的逻辑。

具体来看，以太坊账户分为两种类型：合约账户（Contracts Accounts）和外部账户（Externally Owned Accounts，或 EOA）。

- 合约账户：存储执行的智能合约代码，只能被外部账户来调用激活；
- 外部账户：以太币拥有者账户，对应到某公钥。账户包括 nonce、balance、storageRoot、codeHash 等字段，由个人来控制。

当合约账户被调用时，存储其中的智能合约会在矿工处的虚拟机中自动执行，并消耗一定的燃料。燃料通过外部账户中的以太币进行购买。

交易

交易（Transaction），在以太坊中是指从一个账户到另一个账户的消息数据。消息数据可以是以太币或者合约执行参数。

以太坊采用交易作为执行操作的最小单位。每个交易包括如下字段：

- to：目标账户地址。
- value：可以指定转移的以太币数量。
- nonce：交易相关的字串，用于防止交易被重放。
- gasPrice：执行交易需要消耗的 Gas 价格。
- startgas：交易消耗的最大 Gas 值。
- signature：签名信息。

类似比特币网络，在发送交易时，用户需要缴纳一定的交易费用，通过以太币方式进行支付和消耗。目前，以太坊网络可以支持超过比特币网络的交易速率（可以达到每秒几十笔）。

以太币

以太币（Ether）是以太坊网络中的货币。

以太币主要用于购买燃料，支付给矿工，以维护以太坊网络运行智能合约的费用。以太币最小单位是 wei，一个以太币等于 10^18 个 wei。

以太币同样可以通过挖矿来生成，成功生成新区块的以太坊矿工可以获得 5 个以太币的奖励，以及包含在区块内交易的燃料费用。用户也可以通过交易市场来直接购买以太币。

目前每年大约可以通过挖矿生成超过一千万个以太币，单个以太币的市场价格目前超过 300 美金。

燃料

燃料（Gas），控制某次交易执行指令的上限。每执行一条合约指令会消耗固定的燃料。当某个交易还未执行结束，而燃料消耗完时，合约执行终止并回滚状态。

Gas 可以跟以太币进行兑换。需要注意的是，以太币的价格是波动的，但运行某段智能合约的燃料费用可以是固定的，通过设定 Gas 价格等进行调节。

主要设计

以太坊项目的基本设计与比特币网络类似。为了支持更复杂的智能合约，以太坊在不少地方进行了改进，包括交易模型、共识、对攻击的防护和可扩展性等。

智能合约相关设计

运行环境

以太坊采用以太坊虚拟机作为智能合约的运行环境。以太坊虚拟机是一个隔离的轻量级虚拟机环境，运行在其中的智能合约代码无法访问本地网络、文件系统或其它进程。

对同一个智能合约来说，往往需要在多个以太坊虚拟机中同时运行多份，以确保整个区块链数据的一致性和高度的容错性。另一方面，这也限制了整个网络的容量。

开发语言

以太坊为编写智能合约设计了图灵完备的高级编程语言，降低了智能合约开发的难度。

目前 Solidity 是最常用的以太坊合约编写语言之一。

智能合约编写完毕后，用编译器编译为以太坊虚拟机专用的二进制格式（EVM bytecode），由客户端上传到区块链当中，之后在矿工的以太坊虚拟机中执行。

交易模型

出于智能合约的便利考虑，以太坊采用了账户的模型，状态可以实时的保存到账户里，而无需像比特币的 UXTO 模型那样去回溯整个历史。

UXTO 模型和账户模型的对比如下。

特性	**UXTO 模型**	账户模型
状态查询和变更	需要回溯历史	直接访问
存储空间	较大	较小
易用性	较难处理	易于理解和编程
安全性	较好	需要处理好重放攻击等情况
可追溯性	支持历史	不支持追溯历史

共识

以太坊目前采用了基于成熟的 PoW 共识的变种算法 Ethash 协议作为共识机制。

为了防止 ASIC 矿机矿池的算力攻击，跟原始 PoW 的计算密集型 Hash 运算不同，Ethash 在执行时候需要消耗大量内存，反而跟计算效率关系不大。这意味着很难制造出专门针对 Ethash 的芯片，反而是通用机器可能更加有效。

虽然，Ethash 相对原始的 PoW 进行了改进，但仍然需要进行大量无效的运算，这也为人们所诟病。

社区已经有计划在未来采用更高效的 Proof-of-Stake（PoS）作为共识机制。相对 PoW 机制来讲，PoS 机制无需消耗大量无用的 Hash 计算，但其共识过程的复杂度要更高一些，还有待进一步的检验。

降低攻击

以太坊网络中的交易更加多样化，也就更容易受到攻击。

以太坊网络在降低攻击方面的核心设计思想，仍然是通过经济激励机制防止少数人作恶：

- 所有交易都要提供交易费用，避免 DDoS 攻击；
- 程序运行指令数通过 Gas 来限制，所消耗的费用超过设定上限时就会被取消，避免出现恶意合约。

这就确保了攻击者试图消耗网络中虚拟机的计算资源时，需要付出经济代价（支付大量的以太币）；同时难以通过构造恶意的循环或不稳定合约代码来对网络造成破坏。

提高扩展性

可扩展性是以太坊网络承接更多业务量的最大制约。

以太坊项目未来希望通过分片（sharding）机制来提高整个网络的扩展性。

分片是一组维护和执行同一批智能合约的节点组成的子网络，是整个网络的子集。

支持分片功能之前，以太坊整个网络中的每个节点都需要处理所有的智能合约，这就造成了网络的最大处理能力会受限于单个节点的处理能力。

分片后，同一片内的合约处理是同步的，彼此达成共识，不同分片之间则可以是异步的，可以提高网络整体的可扩展性。

相关工具

客户端和开发库

以太坊客户端可用于接入以太坊网络，进行账户管理、交易、挖矿、智能合约等各方面操作。

以太坊社区现在提供了多种语言实现的客户端和开发库，支持标准的 JSON-RPC 协议。用户可根据自己熟悉的开发语言进行选择。

- go-ethereum：Go 语言实现；
- Parity：Rust 语言实现；
- cpp-ethereum：C++ 语言实现；
- ethereumjs-lib：javascript 语言实现；
- Ethereum(J)：Java 语言实现；
- ethereumH：Haskell 语言实现；
- pyethapp：Python 语言实现；
- ruby-ethereum：Ruby 语言实现。

Geth

上述实现中，go-ethereum 的独立客户端 Geth 是最常用的以太坊客户端之一。

用户可通过安装 Geth 来接入以太坊网络并成为一个完整节点。Geth 也可作为一个 HTTP-RPC 服务器，对外暴露 JSON-RPC 接口，供用户与以太坊网络交互。

Geth 的使用需要基本的命令行基础，其功能相对完整，源码托管于 github.com/ethereum/go-ethereum。

以太坊钱包

对于只需进行账户管理、以太坊转账、DApp 使用等基本操作的用户，则可选择直观易用的钱包客户端。

Mist 是官方提供的一套包含图形界面的钱包客户端，除了可用于进行交易，也支持直接编写和部署智能合约。

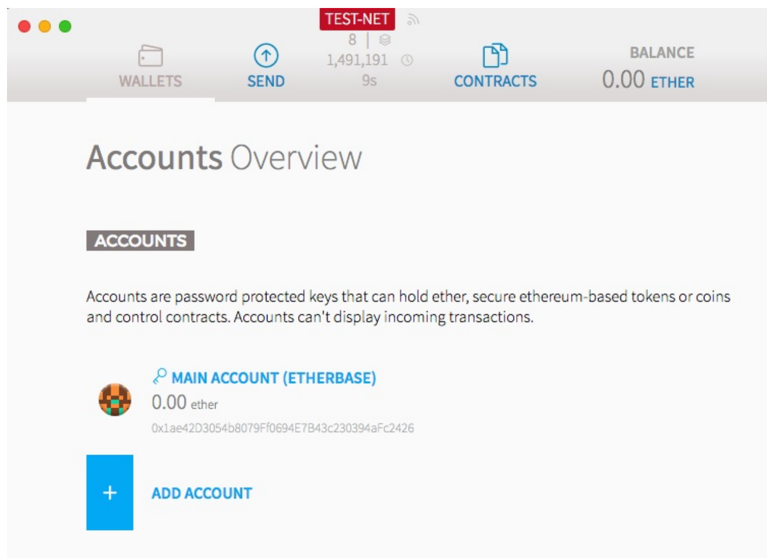


图 *1.10.4.1 - Mist* 浏览器

所编写的代码编译发布后，可以部署到区块链上。使用者可通过发送调用相应合约方法的交易，来执行智能合约。

IDE

对于开发者，以太坊社区涌现出许多服务于编写智能合约和 DApp 的 IDE，例如：

- Truffle：一个功能丰富的以太坊应用开发环境。
- Embark：一个 DApp 开发框架，支持集成以太坊、IPFS 等。
- Remix：一个用于编写 Solidity 的 IDE，内置调试器和测试环境。

网站资源

已有一些网站提供对以太坊网络的数据、运行在以太坊上的 DApp 等信息进行查看，例如：

- ethstats.net：实时查看网络的信息，如区块、价格、交易数等。
- ethernodes.org：显示整个网络的历史统计信息，如客户端的分布情况等。
- dapps.ethercasts.com：查看运行在以太坊上的 DApp 的信息，包括简介、所处阶段和状态等。

图 *1.10.4.2 - *以太坊网络上的 *Dapp* 信息

安装客户端

本节将介绍如何安装 Geth，即 Go 语言实现的以太坊客户端。这里以 Ubuntu 16.04 操作系统为例，介绍从 PPA 仓库和从源码编译这两种方式来进行安装。

从 PPA 直接安装

首先安装必要的工具包。

```
$ apt-get install software-properties-common
```

之后用以下命令添加以太坊的源。

```
$ add-apt-repository -y ppa:ethereum/ethereum
$ apt-get update
```

最后安装 go-ethereum。

```
$ apt-get install ethereum
```

安装成功后，则可以开始使用命令行客户端 Geth。可用 `geth --help` 查看各命令和选项，例如，用以下命令可查看 Geth 版本为 1.6.1-stable。

```
$ geth version

Geth
Version: 1.6.1-stable
Git Commit: 021c3c281629baf2eae967dc2f0a7532ddfdc1fb
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.8.1
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.8
```

从源码编译

也可以选择从源码进行编译安装。

安装 Go 语言环境

Go 语言环境可以自行访问 golang.org 网站下载二进制压缩包安装。注意不推荐通过包管理器安装版本，往往比较旧。

如下载 Go 1.8 版本，可以采用如下命令。

```
$ curl -O https://storage.googleapis.com/golang/go1.8.linux-amd64.tar.gz
```

下载完成后，解压目录，并移动到合适的位置（推荐为 /usr/local 下）。

```
$ tar -xvf go1.8.linux-amd64.tar.gz
$ sudo mv go /usr/local
```

安装完成后记得配置 GOPATH 环境变量。

```
$ export GOPATH=YOUR_LOCAL_GO_PATH/Go
$ export PATH=$PATH:/usr/local/go/bin:$GOPATH/bin
```

此时，可以通过 `go version` 命令验证安装 是否成功。

```
$ go version

go version go1.8 linux/amd64
```

下载和编译 Geth

用以下命令安装 C 的编译器。

```
$ apt-get install -y build-essential
```

下载选定的 go-ethereum 源码版本，如最新的社区版本：

```
$ git clone https://github.com/ethereum/go-ethereum
```

编译安装 Geth。

```
$ cd go-ethereum
$ make geth
```

安装成功后，可用 `build/bin/geth --help` 查看各命令和选项。例如，用以下命令可查看 Geth 版本为 1.6.3-unstable。

```
$ build/bin/geth version
Geth
Version: 1.6.3-unstable
Git Commit: 067dc2cbf5121541aea8c6089ac42ce07582ead1
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.8
Operating System: linux
GOPATH=/usr/local/gopath/
GOROOT=/usr/local/go
```

使用智能合约

以太坊社区有不少提供智能合约编写、编译、发布、调用等功能的工具，用户和开发者可以根据需求或开发环境自行选择。

本节将向开发者介绍使用 Geth 客户端搭建测试用的本地区块链，以及如何在链上部署和调用智能合约。

搭建测试用区块链

由于在以太坊公链上测试智能合约需要消耗以太币，所以对于开发者开发测试场景，可以选择本地自行搭建一条测试链。开发好的智能合约可以很容易的切换接口部署到公有链上。注意测试链不同于以太坊公链，需要给出一些非默认的手动配置。

配置初始状态

首先配置私有区块链网络的初始状态。新建文件 `genesis.json` ，内容如下。

```
{
  "config": {
        "chainId": 22,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
  },
  "alloc"      : {},
  "coinbase"   : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x400",
  "extraData"  : "",
  "gasLimit"   : "0x2fefd8",
  "nonce"      : "0x0000000000000038",
  "mixhash"    : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp"  : "0x00"
}
```

其中， `chainId` 指定了独立的区块链网络 ID，不同 ID 网络的节点无法互相连接。配置文件还对当前挖矿难度 `difficulty` 、区块 Gas 消耗限制 `gasLimit` 等参数进行了设置。

启动区块链

用以下命令初始化区块链，生成创世区块和初始状态。

```
$ geth --datadir /path/to/datadir init /path/to/genesis.json
```

其中，`--datadir` 指定区块链数据的存储位置，可自行选择一个目录地址。

接下来用以下命令启动节点，并进入 Geth 命令行界面。

```
$ geth --identity "TestNode" --rpc --rpcport "8545" --datadir /path/to/datadir --port
"30303" --nodiscover console
```

各选项的含义如下。

- `--identity` ：指定节点 ID；
- `--rpc` ： 表示开启 HTTP-RPC 服务；
- `--rpcport` ： 指定 HTTP-RPC 服务监听端口号（默认为 8545）；
- `--datadir` ： 指定区块链数据的存储位置；
- `--port` ： 指定和其他节点连接所用的端口号（默认为 30303）；
- `--nodiscover` ： 关闭节点发现机制，防止加入有同样初始配置的陌生节点；

创建账号

用上述 `geth console` 命令进入的命令行界面采用 JavaScript 语法。可以用以下命令新建一个账号。

```
> personal.newAccount()

Passphrase:
Repeat passphrase:
"0x1b6eaa5c016af9a3d7549c8679966311183f129e"
```

输入两遍密码后，会显示生成的账号，如 `"0x1b6eaa5c016af9a3d7549c8679966311183f129e"` 。可以用以下命令查看该账号余额。

```
> myAddress = "0x1b6eaa5c016af9a3d7549c8679966311183f129e"
> eth.getBalance(myAddress)
0
```

看到该账号当前余额为 0。可用 `miner.start()` 命令进行挖矿，由于初始难度设置的较小，所以很容易就可挖出一些余额。`miner.stop()` 命令可以停止挖矿。

创建和编译智能合约

以 Solidity 编写的智能合约为例。为了将合约代码编译为 EVM 二进制，需要安装 Solidity 编译器 solc。

```
$ apt-get install solc
```

新建一个 Solidity 智能合约文件，命名为 `testContract.sol`，内容如下。该合约包含一个方法 multiply，作用是将输入的整数乘以 7 后输出。

```
pragma solidity ^0.4.0;
contract testContract {
  function multiply(uint a) returns(uint d) {
    d = a * 7;
  }
}
```

用 solc 获得合约编译后的 EVM 二进制码。

```
$ solc --bin testContract.sol

======= testContract.sol:testContract =======
Binary:
60606040523415610000c57fe5b5b60a58061001b6000396000f30060606040526000357c01000000000000
00000000000000000000000000000000000000000000900463ffffffff168063c6888fa114603a575bfe5b
3415604157fe5b6055600480803590602001909190505060606b565b6040518082815260200191505060604051
80910390f35b6000600782029050505b9190505600a165627a7a72305820748467daab52f2f1a63180df2c49
26f3431a2aa82dcdfbcbde5e7d036742a94b0029
```

再用 solc 获得合约的 JSON ABI（Application Binary Interface），其中指定了合约接口，包括可调用的合约方法、变量、事件等。

```
$ solc --abi testContract.sol

======= testContract.sol:testContract =======
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"a","type":"uint256"}],"name":"multiply","outputs"
:[{"name":"d","type":"uint256"}],"payable":false,"type":"function"}]
```

下面回到 Geth 的 JavaScript 环境命令行界面，用变量记录上述两个值。注意在 code 前加上 `0x` 前缀。

```
> code = "0x6060604052341561000c57fe5b5b60a58061001b6000396000f30060606040526000357c01
00000000000000000000000000000000000000000000000000000000900463ffffffff168063c6888fa114
603a575bfe5b3415604157fe5b6055600480803590602001909190505060606b565b60405180828152602001
91505060405180910390f35b6000600782029050b9190505600a165627a7a72305820748467daab52f2f1
a63180df2c4926f3431a2aa82dcdfbcbde5e7d036742a94b0029"
> abi = [{"constant":false,"inputs":[{"name":"a","type":"uint256"}],"name":"multiply",
"outputs":[{"name":"d","type":"uint256"}],"payable":false,"type":"function"}]
```

部署智能合约

在 Geth 的 JavaScript 环境命令行界面，首先用以下命令解锁自己的账户，否则无法发送交易。

```
> personal.unlockAccount(myAddress)

Unlock account 0x1b6eaa5c016af9a3d7549c8679966311183f129e
Passphrase:
true
```

接下来发送部署合约的交易。

```
> myContract = eth.contract(abi)
> contract = myContract.new({from:myAddress,data:code,gas:1000000})
```

如果此时没有在挖矿，用 `txpool.status` 命令可看到本地交易池中有一个待确认的交易。可用以下命令查看当前待确认的交易。

```
> eth.getBlock("pending",true).transactions

[{
    blockHash: "0xbf0619ca48d9e3cc27cd0ab0b433a49a2b1bed90ab57c0357071b033aca1f2cf",
    blockNumber: 17,
    from: "0x1b6eaa5c016af9a3d7549c8679966311183f129e",
    gas: 90000,
    gasPrice: 20000000000,
    hash: "0xa019c2e5367b3ad2bbfa427b046ab65c81ce2590672a512cc973b84610eee53e",
    input: "0x6060604052341561000c57fe5b5b60a58061001b6000396000f30060606040526000357c
0100000000000000000000000000000000000000000000000000000000900463ffffffff168063c6888fa1
14603a575bfe5b3415604157fe5b6055600480803590602001909190505060606b565b60405180828152602 0
0191505060405180910390f35b60006007820290505b9190505600a165627a7a72305820748467daab52f2
f1a63180df2c4926f3431a2aa82dcdfbcbde5e7d036742a94b0029",
    nonce: 1,
    r: "0xbcb2ba94f45dfb900a0533be3c2c603c2b358774e5fe89f3344031b202995a41",
    s: "0x5f55fb1f76aa11953e12746bc2d19fbea6aeb1b9f9f1c53a2eefab7058515d99",
    to: null,
    transactionIndex: 0,
    v: "0x4f",
    value: 0
}]
```

可以用 `miner.start()` 命令挖矿，一段时间后，交易会被确认，即随新区块进入区块链。

调用智能合约

用以下命令可以发送交易，其中 sendTransaction 方法的前几个参数与合约中 multiply 方法的输入参数对应。这种方式，交易会通过挖矿记录到区块链中，如果涉及状态改变也会获得全网共识。

```
> contract.multiply.sendTransaction(10, {from:myAddress})
```

如果只是想本地运行该方法查看返回结果，可采用如下方式获取结果。

```
> contract.multiply.call(10)
70
```

智能合约案例：投票

本节将介绍一个用 Solidity 语言编写的智能合约案例。代码来源于 Solidity 官方文档 中的示例。

该智能合约实现了一个自动化的、透明的投票应用。投票发起人可以发起投票，将投票权赋予投票人；投票人可以自己投票，或将自己的票委托给其他投票人；任何人都可以公开查询投票的结果。

智能合约代码

实现上述功能的合约代码如下所示，并不复杂，语法跟 JavaScript 十分类似。

```
pragma solidity ^0.4.11;

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    struct Proposal {
        bytes32 name;
        uint voteCount;
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    // Create a new ballot to choose one of `proposalNames`
    function Ballot(bytes32[] proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) {
        require((msg.sender == chairperson) && !voters[voter].voted);
```

```
        voters[voter].weight = 1;
    }

    // Delegate your vote to the voter `to`.
    function delegate(address to) {
        Voter sender = voters[msg.sender];
        require(!sender.voted);
        require(to != msg.sender);

        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender);
        }

        sender.voted = true;
        sender.delegate = to;
        Voter delegate = voters[to];
        if (delegate.voted) {
            proposals[delegate.vote].voteCount += sender.weight;
        } else {
            delegate.weight += sender.weight;
        }
    }

    // Give your vote (including votes delegated to you)
    // to proposal `proposals[proposal].name`.
    function vote(uint proposal) {
        Voter sender = voters[msg.sender];
        require(!sender.voted);
        sender.voted = true;
        sender.vote = proposal;

        proposals[proposal].voteCount += sender.weight;
    }

    // @dev Computes the winning proposal taking all
    // previous votes into account.
    function winningProposal() constant
            returns (uint winningProposal)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() constant
            returns (bytes32 winnerName)
```

```
        {
            winnerName = proposals[winningProposal()].name;
        }
    }
```

代码解析

指定版本

在第一行，`pragma` 关键字指定了和该合约兼容的编译器版本。

```
pragma solidity ^0.4.11;
```

该合约指定，不兼容比 `0.4.11` 更旧的编译器版本，且 `^` 符号表示也不兼容从 `0.5.0` 起的新编译器版本。即兼容版本范围是 `0.4.11 <= version < 0.5.0`。该语法与 npm 的版本描述语法一致。

结构体类型

Solidity 中的合约（contract）类似面向对象编程语言中的类。每个合约可以包含状态变量、函数、事件、结构体类型和枚举类型等。一个合约也可以继承另一个合约。

在本例命名为 `Ballot` 的合约中，声明了 2 个结构体类型：`Voter` 和 `Proposal`。

- `struct Voter`：投票人，其属性包括 `uint weight`（该投票人的权重）、`bool voted`（是否已投票）、`address delegate`（如果该投票人将投票委托给他人，则记录受委托人的账户地址）和 `uint vote`（投票做出的选择，即相应提案的索引号）。
- `struct Proposal`：提案，其属性包括 `bytes32 name`（名称）和 `uint voteCount`（已获得的票数）。

需要注意，`address` 类型记录了一个以太坊账户的地址。`address` 可看作一个数值类型，但也包括一些与以太币相关的方法，如查询余额 `<address>.balance`、向该地址转账 `<address>.transfer(uint256 amount)` 等。

状态变量

合约中的状态变量会长期保存在区块链中。通过调用合约中的函数，这些状态变量可以被读取和改写。

本例中定义了 3 个状态变量：`chairperson`、`voters`、`proposals`。

- `address public chairperson`：投票发起人，类型为 `address`。
- `mapping(address => Voter) public voters`：所有投票人，类型为 `address` 到 `Voter` 的

映射。

- `Proposal[] public proposals` ：所有提案，类型为动态大小的 `Proposal` 数组。

3 个状态变量都使用了 `public` 关键字，使得变量可以被外部访问（即通过消息调用）。事实上，编译器会自动为 `public` 的变量创建同名的 getter 函数，供外部直接读取。

状态变量还可设置为 `internal` 或 `private` 。`internal` 的状态变量只能被该合约和继承该合约的子合约访问，`private` 的状态变量只能被该合约访问。状态变量默认为 `internal` 。

将上述关键状态信息设置为 `public` 能够增加投票的公平性和透明性。

函数

合约中的函数用于处理业务逻辑。函数的可见性默认为 `public` ，即可以从内部或外部调用，是合约的对外接口。函数可见性也可设置为 `external` 、`internal` 和 `private` 。

本例实现了 6 个 `public` 函数，可看作 6 个对外接口，功能分别如下。

创建投票

函数 `function Ballot(bytes32[] proposalNames)` 用于创建一个新的投票。

所有提案的名称通过参数 `bytes32[] proposalNames` 传入，逐个记录到状态变量 `proposals` 中。同时用 `msg.sender` 获取当前调用消息的发送者的地址，记录为投票发起人 `chairperson` ，该发起人投票权重设为 1。

赋予投票权

函数 `function giveRightToVote(address voter)` 实现给投票人赋予投票权。

该函数给 `address voter` 赋予投票权，即将 `voter` 的投票权重设为 1，存入 `voters` 状态变量。

这个函数只有投票发起人 `chairperson` 可以调用。这里用到了 `require((msg.sender == chairperson) && !voters[voter].voted)` 函数。如果 `require` 中表达式结果为 `false` ，这次调用会中止，且回滚所有状态和以太币余额的改变到调用前。但已消耗的 Gas 不会返还。

委托投票权

函数 `function delegate(address to)` 把投票委托给其他投票人。

其中，用 `voters[msg.sender]` 获取委托人，即此次调用的发起人。用 `require` 确保发起人没有投过票，且不是委托给自己。由于被委托人也可能已将投票委托出去，所以接下来，用 `while` 循环查找最终的投票代表。找到后，如果投票代表已投票，则将委托人的权重加到所投的提案上；如果投票代表还未投票，则将委托人的权重加到代表的权重上。

该函数使用了 `while` 循环，这里合约编写者需要十分谨慎，防止调用者消耗过多 Gas，甚至出现死循环。

进行投票

函数 `function vote(uint proposal)` 实现投票过程。

其中，用 `voters[msg.sender]` 获取投票人，即此次调用的发起人。接下来检查是否是重复投票，如果不是，进行投票后相关状态变量的更新。

查询获胜提案

函数 `function winningProposal() constant returns (uint winningProposal)` 将返回获胜提案的索引号。

这里，`returns (uint winningProposal)` 指定了函数的返回值类型，`constant` 表示该函数不会改变合约状态变量的值。

函数通过遍历所有提案进行记票，得到获胜提案。

查询获胜者名称

函数 `function winnerName() constant returns (bytes32 winnerName)` 实现返回获胜者的名称。

这里采用内部调用 `winningProposal()` 函数的方式获得获胜提案。如果需要采用外部调用，则需要写为 `this.winningProposal()` 。

本章小结

以太坊项目将区块链技术在数字货币的基础上进行了延伸，提出打造更为通用的智能合约平台的宏大构想，并基于开源技术构建了以太坊为核心的开源生态系统。

本章内容介绍了以太坊的相关知识，包括核心概念、设计、工具，以及客户端的安装、智能合约的使用和编写等。

比照比特币项目，读者通过学习可以掌握以太坊的相关改进设计，并学习智能合约的编写。实际上，智能合约并不是一个新兴概念，但区块链技术的出现为智能合约的“代码即律法”提供提供了信任基础和实施架构。通过引入智能合约，区块链技术释放了支持更多应用领域的巨大潜力。

Hyperledger - 超级账本项目

Uneasy lies the head that wears a crown.

Hyperledger 项目是首个面向企业的开放区块链技术的重要探索。在 Linux 基金会的支持下，吸引了包括 IBM、Intel、摩根等在内的众多科技和金融巨头的参与。

本章将介绍 Hyperledger 项目的历史，并以核心的 fabric 项目为例，讲解如何快速安装部署和应用区块链系统。

简介

历史

区块链已经成为当下最受人关注的开源技术，有人说它将颠覆金融行业的未来。然而对很多人来说，区块链技术难以理解和实现，而且缺乏统一的规范。

2015 年 12 月，开源世界的旗舰——Linux 基金会牵头，联合 30 家初始企业成员（包括 IBM、Accenture、Intel、J.P.Morgan、R3、DAH、DTCC、FUJITSU、HITACHI、SWIFT、Cisco 等），共同 宣告 了 Hyperledger 项目的成立。该项目试图打造一个透明、公开、去中心化的分布式账本项目，作为区块链技术的开源规范和标准，让更多的应用能更容易的建立在区块链技术之上。项目官方信息网站在 hyperledger.org，

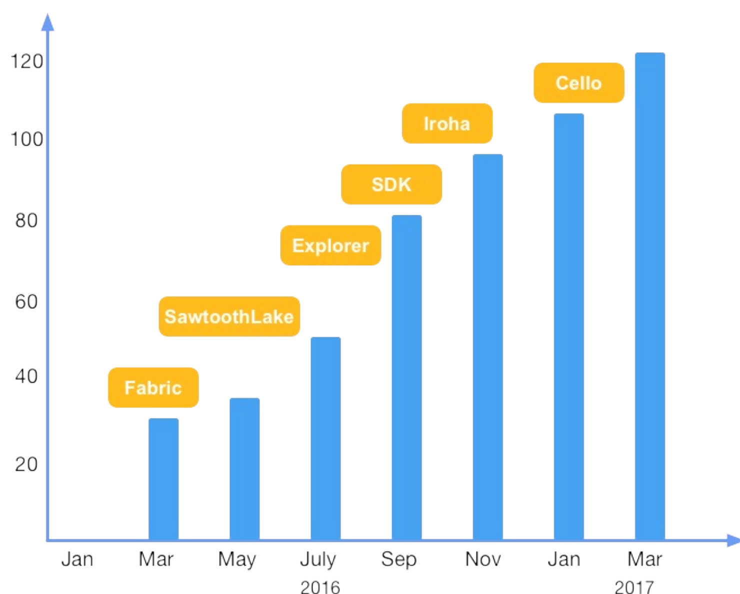


图 *1.11.1.1 - Hyperledger* 项目快速成长

目前已经有超过 120 家全球知名企业和机构（大部分均为各自行业的领导者）宣布加入 Hyperledger 项目，其中包括 30 家来自中国本土的企业，包括艾亿新融旗下的艾亿数融科技公司（2016.05.19）、Onchain（2016.06.22）、比邻共赢（Belink）信息技术有限公司（2016.06.22）、BitSE（2016.06.22）、布比（2016.07.27）、三一重工（2016.08.30）、万达金融（2016.09.08）、华为（2016.10.24）等。

如果说以比特币为代表的货币区块链技术为 1.0，以以太坊为代表的合同区块链技术为 2.0，那么实现了完备的权限控制和安全保障的 Hyperledger 项目毫无疑问代表着 3.0 时代的到来。

IBM 贡献了数万行已有的 Open Blockchain 代码，Digital Asset 则贡献了企业和开发者相关资源，R3 贡献了新的金融交易架构，Intel 也刚贡献了跟分布式账本相关的代码。

Hyperledger 社区由技术委员会（Technical Steering Committee，TSC）指导，首任主席由来自 IBM 开源技术部 CTO 的 Chris Ferris 担任，管理组主席则由来自 Digital Asset Holdings 的 CEO Blythe Masters 担任。另外，自 2016 年 5 月起，Apache 基金会创始人 Brian Behlendorf 担任超级账本项目的首位执行董事。2016 年 12 月，中国技术工作组 正式成立，负责本土社区组织和技术引导工作。官方网站也提供了十分详细的 组织结构信息。

该项目的出现，实际上宣布区块链技术已经不再是仅面向“社会实验”性质的应用场景，它已经正式被主流机构和企业市场认可；同时，Hyperledger 首次提出和实现的完备权限管理、创新的一致性算法和可拔插、可扩展的框架，对于区块链相关技术和产业的发展都将产生深远的影响。

主要项目

代码托管在 Gerrit 和 Github（自动从 gerrit 上同步）上。

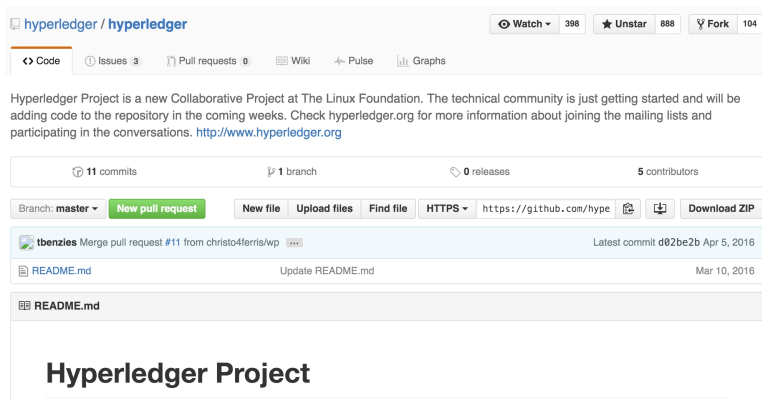


图 *1.11.1.2 - Hyperledger*

目前主要包括三大账本平台项目和若干其它项目。

账本平台项目：

- Fabric：包括 Fabric、Fabric CA、Fabric SDK（包括 Node.Js、Python 和 Java 等语言）和 fabric-api、fabric-sdk-node、fabric-sdk-py 等，目标是区块链的基础核心平台，支持 pbft 等新的 consensus 机制，支持权限管理，最早由 IBM 和 DAH 发起；
- SawToothLake：包括 arcade、core、dev-tools、validator、mktplace 等。是 Intel 主要发起和贡献的区块链平台，支持全新的基于硬件芯片的共识机制 Proof of Elapsed Time（PoET）。
- Iroha：账本平台项目，基于 C++ 实现，带有不少面向 Web 和 Mobile 的特性，主要由 Soramitsu 发起和贡献。

其它项目：

- Blockchain Explorer：提供 Web 操作界面，通过界面快速查看查询绑定区块链的状态（区块个数、交易历史）信息等。
- Cello：提供"Blockchain as a Service" 功能，使用 Cello，管理员可以轻松获取和管理多条区块链；应用开发者可以无需关心如何搭建和维护区块链。

目前，所有项目均处于孵化（Incubation）状态。

项目原则

项目约定共同遵守的 基本原则 为：

- 重视模块化设计，包括交易、合同、一致性、身份、存储等技术场景；
- 代码可读性，保障新功能和模块都可以很容易添加和扩展；
- 演化路线，随着需求的深入和更多的应用场景，不断增加和演化新的项目。

如果你对 Hyperledger 的源码实现感兴趣，可以参考 Hyperledger 源码分析之 Fabric。

社区组织

任何一个成功的开源项目，都离不开一个健康繁荣的社区。

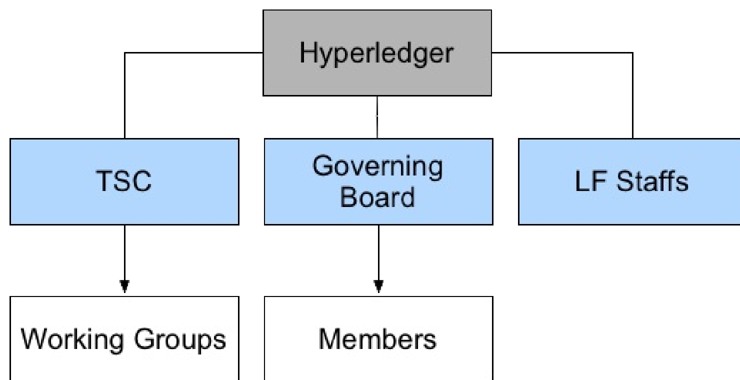


图 *1.11.2.1 - Hyperledger* 社区组织结构

社区目前主要是三驾马车领导的结构：

- Technical Steering Committee （技术委员会）：负责技术相关的工作，下设多个工作组，具体带动各个项目的发展；
- Governing Board （管理董事会）：负责社区组织的整体决策，由 Hyperledger 会员中推选出代表；
- Linux Foundation（Linux 基金会）：协助 Hyperledger 社区在 Linux 基金会的支持下发展。

中国技术工作组

随着开源精神和开源文化在中国的普及，越来越多的企业和组织开始意识到共同构建一个健康的生态系统的重要性，也愿意为开源事业做出一定贡献。

Linux 基金会和 Hyperledger 社区十分重视项目在大中华区的应用和落地情况，并希望能为中国技术人员贡献开源社区提供方便。在此背景下，Hyperledger 执行董事 Brian Behlendorf 于 2016 年 12 月 1 日 提议 成立 中国技术工作组，即 TWG-China，并得到了 TSC 的一致支持通过。

工作组的 主要职责 包括：

- 带领和引导大中华区的技术相关活动，包括贡献代码、指南文档、项目提案等。
- 推动技术相关的交流、会员之间的合作和实践案例的落地；
- 通过邮件列表、rocketchat、论坛等方式提供会员的技术交流；

- 协助举办社区活动，包括 meetup、黑客松、hackfest、技术分享、培训等。

目前，工作组由来自 IBM、万达、华为等 Hyperledger 成员企业的技术专家领导，并得到了来自社区的众多志愿者的支持。

开发和提交代码

安装环境

推荐在 Linux（如 Ubuntu 14.04+）或 MacOS 环境中开发代码，并安装如下工具。

- git：用来获取代码。
- golang 1.6+：安装成功后需要配置 $GOPATH 等环境变量。
- Docker 1.12+：用来支持容器环境，注意 MacOS 下要用 Docker for Mac。

获取代码

首先注册 Linux foundation ID，并登陆 https://gerrit.hyperledger.org/，添加个人 ssh pub key。

查看项目列表，找到对应项目，以 fabric 为例，采用 `Clone with commit-msg hook` 的方式来获取。

典型的，执行如下命令获取代码，放到 `$GOPATH/src/github.com/hyperledger/` 路径下，其中 `LF_ID` 替换为你的 Linux foundation id。

```
$ mkdir $GOPATH/src/github.com/hyperledger/
$ cd $GOPATH/src/github.com/hyperledger/
$ git clone ssh://LF_ID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418 LF_ID@g
errit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

如果没有添加个人 ssh pubkey，则可以通过 https 方式 clone，需要输入用户名和密码信息。

```
git clone http://LF_ID@gerrit.hyperledger.org/r/fabric && (cd fabric && curl -kLo `git
 rev-parse --git-dir`/hooks/commit-msg http://LF_ID@gerrit.hyperledger.org/r/tools/hoo
ks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)
```

编译和测试

大部分编译和安装过程都可以利用 Makefile 来执行，包括如下常见操作。

安装 go tools

执行

```
$ make gotools
```

语法格式检查

执行

```
$ make linter
```

编译 peer

执行

```
$ make peer
```

会自动编译生成 Docker 镜像，并生成本地 peer 可执行文件。

注意：有时候会因为获取安装包不稳定而报错，需要执行 `make clean`，然后再次执行。

生成 Docker 镜像

执行

```
$ make images
```

执行所有的检查和测试

执行

```
$ make checks
```

执行单元测试

执行

```
$ make unit-test
```

如果要运行某个特定单元测试，则可以通过类似如下格式。

```
$ go test -v -run=TestGetFoo
```

执行 BDD 测试

需先生成本地 Docker 镜像。

执行

```
$ make behave
```

提交代码

仍然使用 Linux foundation ID 登录 jira.hyperledger.org，查看有没有未分配的任务，如果对某个任务感兴趣，可以添加自己为 assignee，如对 FAB-XXX 任务。

本地创建新的分支 FAB-XXX。

```
$ git checkout -b FAB-XXX
```

实现任务代码，完成后，执行语法格式检查和测试等，确保所有检查和测试都通过。

提交代码到本地仓库。

```
$ git commit -a -s
```

会打开一个窗口需要填写 commit 信息，格式一般要求为：

```
Simple words to describe main change

This fixes #FAB-XXX.

A more detailed description can be here, with several
paragraphs and sentences...
```

之后使用 git review 命令推送到远端仓库。

```
$ git review
```

提交成功后，可以打开 gerrit.hyperledger.org/r/，查看自己最新提交的 patchset 信息，添加几位 reviewer。之后就是等待开发者团队的 review 结果，如果得到通过，则会被项目的 maintainer 们 merge 到主分支。否则还需要针对大家提出的建议进一步的修正。

修正过程跟提交代码过程类似，唯一不同是提交的时候使用

```
$ git commit -a --amend
```

表示这个提交是对旧提交的一次修订。

完整流程

总结，完整的流程如下图所示。

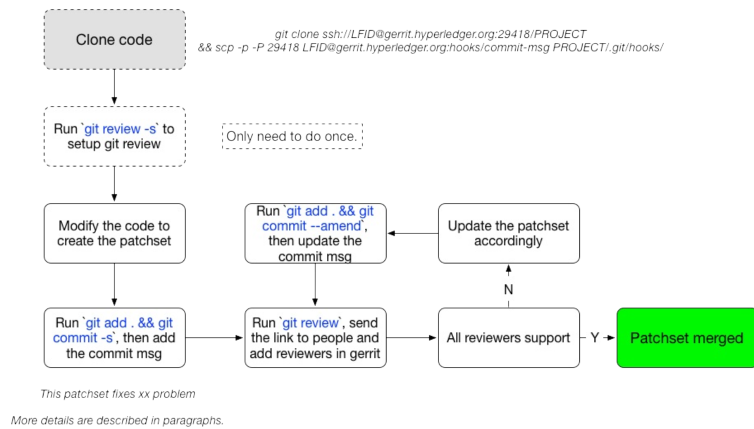


图 *1.11.4.1* - 代码提交流程

小结

Hyperledger 是 Linux 基金会支持的分布式账本平台，这是开源界试图构建一套标准化分布式账本平台的重要尝试。

类似的项目还包括 以太坊平台、R3 CEV 牵头的 Corda 项目、微软的 bletchley 项目 等。

Fabric 部署与管理

简介

使用 1.0 版本

Hyperledger Fabric 1.0 版本整体 重新设计了架构，新的设计可以实现更好的扩展性和安全性。

下载 Compose 模板文件

```
$ git clone https://github.com/yeasy/docker-compose-files
```

进入 `hyperledger/1.0` 目录，查看包括若干模板文件，功能如下。

文件	功能
orderer-base.yaml	orderer 节点的基础服务模板
peer-base.yaml	peer 节点的基础服务模板
docker-compose-base.yaml	包含 orderer 和 peers 组织结构的基础服务模板
docker-compose-1peer.yaml	使用自定义的 `channel` 启动一个最小化的环境，包括 1 个 peer 节点、1 个 orderer 节点、1 个 CA 节点、1 个 cli 节点
docker-compose-2orgs-4peers.yaml	使用自定义的 `channel` 启动一个环境，包括 4 个 peer 节点、1 个 orderer 节点、1 个 CA 节点、1 个 cli 节点
docker-compose-2orgs-4peers-couchdb.yaml	启动一个带有 couchdb 服务的网络环境
docker-compose-2orgs-4peers-event.yaml	启动一个带有 event 事件服务的网络环境
e2e_cli/channel-artifacts	存放创建 orderer, channel, anchor peer 操作时的配置文件
e2e_cli/crypto-config	存放 orderer 和 peer 相关证书
e2e_cli/example	用来测试的 chaincode
scripts/setup_Docker.sh	安装并配置 dokcer 和 docker-compose
scripts/download_images.sh	下载依赖镜像
scripts/start_fabric.sh	快速启动一个fabric 网络
scripts/initialize.sh	自动化测试脚本，用来初始化 `channel` 和 `chaincode`
scripts/test_4peers.sh	自动化测试脚本，用来执行 chaincode 操作
scripts/cleanup_env.sh	容器，镜像自动清除脚本
scripts/test_1peer.sh	测试1个peer网络的自动化脚本
kafka/	基于kafka 的 ordering 服务

安装 Docker 和 docker-compose

docker 及 docker-compose 可以自行手动安装。也可以通过 hyperledger/1.0/scripts 提供的 `setup_Docker.sh` 脚本自动安装。

```
$ bash scripts/setup_Docker.sh
```

获取 Docker 镜像

Docker 镜像可以自行从源码编译（`make docker`），或从 DockerHub 仓库下载。

执行脚本获取

直接执行 hyperledger/1.0/scripts 提供的 `download_images.sh` 脚本获取。

```
$ bash scripts/download_images.sh
```

从官方仓库获取

从社区 DockerHub 仓库下载。

```
# pull fabric images
ARCH=x86_64
BASEIMAGE_RELEASE=0.3.1
BASE_VERSION=1.0.0
PROJECT_VERSION=1.0.0
IMG_TAG=1.0.0

echo "Downloading fabric images from DockerHub...with tag = ${IMG_TAG}... need a while"

# TODO: we may need some checking on pulling result?
docker pull hyperledger/fabric-peer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-orderer:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ca:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-tools:$ARCH-$IMG_TAG
docker pull hyperledger/fabric-ccenv:$ARCH-$PROJECT_VERSION
docker pull hyperledger/fabric-baseimage:$ARCH-$BASEIMAGE_RELEASE
docker pull hyperledger/fabric-baseos:$ARCH-$BASEIMAGE_RELEASE

# Only useful for debugging
# docker pull yeasy/hyperledger-fabric

echo "===Re-tagging images to *latest* tag"
docker tag hyperledger/fabric-peer:$ARCH-$IMG_TAG hyperledger/fabric-peer
docker tag hyperledger/fabric-orderer:$ARCH-$IMG_TAG hyperledger/fabric-orderer
docker tag hyperledger/fabric-ca:$ARCH-$IMG_TAG hyperledger/fabric-ca
docker tag hyperledger/fabric-tools:$ARCH-$IMG_TAG hyperledger/fabric-tools
```

从第三方仓库获取

这里也提供了调整（基于 golang:1.8 基础镜像制作）后的第三方镜像，与社区版本功能是一致的。

通过如下命令拉取相关镜像，并更新镜像别名。

```
$ ARCH=x86_64
$ BASEIMAGE_RELEASE=0.3.1
$ BASE_VERSION=1.0.0
$ PROJECT_VERSION=1.0.0
$ IMG_TAG=1.0.0
$ docker pull yeasy/hyperledger-fabric-base:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-peer:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-orderer:$IMG_VERSION \
  && docker pull yeasy/hyperledger-fabric-ca:$IMG_VERSION \
  && docker pull hyperledger/fabric-couchdb:$ARCH-$IMG_VERSION \
  && docker pull hyperledger/fabric-kafka:$ARCH-$IMG_VERSION \
  && docker pull hyperledger/fabric-zookeeper:$ARCH-$IMG_VERSION

$ docker tag yeasy/hyperledger-fabric-peer:$IMG_VERSION hyperledger/fabric-peer \
  && docker tag yeasy/hyperledger-fabric-orderer:$IMG_VERSION hyperledger/fabric-order
er \
  && docker tag yeasy/hyperledger-fabric-ca:$IMG_VERSION hyperledger/fabric-ca \
  && docker tag yeasy/hyperledger-fabric-peer:$IMG_VERSION hyperledger/fabric-tools \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-ccenv:$A
RCH-$PROJECT_VERSION \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-baseos:$
ARCH-$BASEIMAGE_RELEASE \
  && docker tag yeasy/hyperledger-fabric-base:$IMG_VERSION hyperledger/fabric-baseimag
e:$ARCH-$BASEIMAGE_RELEASE \
  && docker tag hyperledger/fabric-couchdb:$ARCH-$IMG_VERSION hyperledger/fabric-couch
db \
  && docker tag hyperledger/fabric-zookeeper:$ARCH-$IMG_VERSION hyperledger/fabric-zoo
keeper \
  && docker tag hyperledger/fabric-kafka:$ARCH-$IMG_VERSION hyperledger/fabric-kafka
```

启动 fabric 1.0 网络

通过如下命令快速启动。

```
$ bash scripts/start_fabric.sh
```

或者

```
$ docker-compose -f docker-compose-2orgs-4peers.yaml up
```

注意输出日志中无错误信息。

此时，系统中包括 7 个容器。

```
$ docker ps -a
CONTAINER ID        IMAGE                           COMMAND                  CREATED
        STATUS              PORTS
                            NAMES
8683435422ca        hyperledger/fabric-peer      "bash -c 'while true;"   19 seconds a
go      Up 18 seconds       7050-7059/tcp
                            fabric-cli
f284c4dd26a0        hyperledger/fabric-peer      "peer node start --pe"   22 seconds a
go      Up 19 seconds       7050/tcp, 0.0.0.0:7051->7051/tcp, 7052/tcp, 7054-7059/tcp,
 0.0.0.0:7053->7053/tcp     peer0.org1.example.com
95fa3614f82c        hyperledger/fabric-ca        "fabric-ca-server sta"   22 seconds a
go      Up 19 seconds       0.0.0.0:7054->7054/tcp
                            fabric-ca
833ca0d8cf41        hyperledger/fabric-orderer   "orderer"                22 seconds a
go      Up 19 seconds       0.0.0.0:7050->7050/tcp
                            orderer.example.com
cd21cfff8298        hyperledger/fabric-peer      "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:9051->7051/tcp,
 0.0.0.0:9053->7053/tcp     peer0.org2.example.com
372b583b3059        hyperledger/fabric-peer      "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:10051->7051/tcp
, 0.0.0.0:10053->7053/tcp   peer1.org2.example.com
47ce30077276        hyperledger/fabric-peer      "peer node start --pe"   22 seconds a
go      Up 20 seconds       7050/tcp, 7052/tcp, 7054-7059/tcp, 0.0.0.0:8051->7051/tcp,
 0.0.0.0:8053->7053/tcp     peer1.org1.example.com
```

测试网络

启动 fabric 网络后，可以进行 chaincode 操作，验证网络是否启动正常。

进入到 cli 容器里面，执行 `initialize.sh` 和 `test_4peers.sh` 脚本。

通过如下命令进入容器 cli 并执行测试脚本。

```
$ docker exec -it fabric-cli bash
$ bash ./scripts/initialize.sh
```

注意输出日志无错误提示，最终返回结果应该为：

```
UTC [main] main -> INFO 00c Exiting.....
===================== Chaincode Instantiation on PEER2 on channel 'businesschannel' is successful =====================


===================== All GOOD, initialization completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

之后同样是在 `cli` 容器里执行 `test_4peers.sh` 脚本

```
$ bash ./scripts/test_4peers.sh
```

输出日志无错误提示，最终返回结果应该为：

```
Query Result: 80
UTC [main] main -> INFO 008 Exiting.....
===================== Query on PEER3 on channel 'businesschannel' is successful =====================

===================== All GOOD, End-2-End execution completed =====================


 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

至此，整个网络启动并验证成功。

Fabric v0.6

Fabric 目前的稳定版本为 v0.6，最新的版本 1.0 还在演进中，即将发布。

v0.6 的架构相对简单，适合作为实验或 PoC 场景使用。

Fabric v0.6 安装部署

如果是初次接触 Hyperledger Fabric 项目，推荐采用如下的步骤，基于 Docker-Compose 的一键部署。

官方文档现在也完善了安装部署的步骤，具体可以参考代码 `doc` 目录下内容。

动手前，建议适当了解一些 *Docker 相关知识*。

安装 Docker

Docker 支持 Linux 常见的发行版，如 Redhat/Centos/Ubuntu 等。

```
$ curl -fsSL https://get.docker.com/ | sh
```

以 Ubuntu 14.04 为例，安装成功后，修改 Docker 服务配置（`/etc/default/docker` 文件）。

```
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --api-cors-header='*'"
```

重启 Docker 服务。

```
$ sudo service docker restart
```

Ubuntu 16.04 中默认采用了 systemd 管理启动服务，Docker 配置文件在 `/etc/systemd/system/docker.service.d/override.conf`。

修改后，需要通过如下命令重启 Docker 服务。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker.service
```

安装 docker-compose

首先，安装 python-pip 软件包。

```
$ sudo aptitude install python-pip
```

安装 docker-compose（推荐为 1.7.0 及以上版本）。

```
$ sudo pip install docker-compose>=1.7.0
```

下载镜像

目前 1.0 代码还没有正式发布，推荐使用 v0.6 分支代码进行测试。

下载相关镜像，并进行配置。

```
$ docker pull yeasy/hyperledger-fabric:0.6-dp \
  && docker pull yeasy/hyperledger-fabric-peer:0.6-dp \
  && docker pull yeasy/hyperledger-fabric-base:0.6-dp \
  && docker pull yeasy/blockchain-explorer:latest \
  && docker tag yeasy/hyperledger-fabric-peer:0.6-dp hyperledger/fabric-peer \
  && docker tag yeasy/hyperledger-fabric-base:0.6-dp hyperledger/fabric-baseimage \
  && docker tag yeasy/hyperledger-fabric:0.6-dp hyperledger/fabric-membersrvc
```

也可以使用 官方仓库 中的镜像。

```
$ docker pull hyperledger/fabric-peer:x86_64-0.6.1-preview \
  && docker pull hyperledger/fabric-membersrvc:x86_64-0.6.1-preview \
  && docker pull yeasy/blockchain-explorer:latest \
  && docker tag hyperledger/fabric-peer:x86_64-0.6.1-preview hyperledger/fabric-peer \
  && docker tag hyperledger/fabric-peer:x86_64-0.6.1-preview hyperledger/fabric-baseim
age \
  && docker tag hyperledger/fabric-membersrvc:x86_64-0.6.1-preview hyperledger/fabric-
membersrvc
```

之后，用户可以选择采用不同的一致性机制，包括 noops、pbft 两类。

使用 noops 模式

noops 默认没有采用 consensus 机制，1 个节点即可，可以用来进行快速测试。

```
$ docker run --name=vp0 \
    --restart=unless-stopped \
    -it \
    -p 7050:7050 \
    -p 7051:7051 \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=vp0 \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_NOOPS_BLOCK_WAIT=10 \
    hyperledger/fabric-peer:latest peer node start
```

使用 PBFT 模式

PBFT 是经典的分布式一致性算法，也是 hyperledger 目前最推荐的算法，该算法至少需要 4 个节点。

首先，下载 Compose 模板文件。

```
$ git clone https://github.com/yeasy/docker-compose-files
```

进入 `hyperledger/0.6/pbft` 目录，查看包括若干模板文件，功能如下。

- `4-peers.yml`：启动 4 个 PBFT peer 节点。
- `4-peers-with-membersrvc.yml`：启动 4 个 PBFT peer 节点 + 1 个 CA 节点，并启用 CA 功能。
- `4-peers-with-explorer.yml`：启动 4 个 PBFT peer 节点 + 1 个 Blockchain-explorer，可以通过 Web 界面监控集群状态。
- `4-peers-with-membersrvc-explorer.yml`：启动 4 个 PBFT peer 节点 + 1 个 CA 节点 + 1 个 Blockchain-explorer，并启用 CA 功能。

例如，快速启动一个 4 个 PBFT 节点的集群。

```
$ docker-compose -f 4-peers.yml up
```

多物理节点部署

上述方案的典型场景是单物理节点上部署多个 Peer 节点。如果要扩展到多物理节点，需要容器云平台的支持，如 Swarm 等。

当然，用户也可以分别在各个物理节点上通过手动启动容器的方案来实现跨主机组网，每个物理节点作为一个 peer 节点。

首先，以 4 节点下的 PBFT 模式为例，配置 4 台互相连通的物理机，分别按照上述步骤配置 Docker，下载镜像。

4 台物理机分别命名为 vp0 ~ vp3。

vp0

vp0 作为初始的探测节点。

```
$ docker run --name=vp0 \
    --net="host" \
    --restart=unless-stopped \
    -it --rm \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=vp0 \
    -e CORE_PBFT_GENERAL_N=4 \
    -e CORE_LOGGING_LEVEL=debug \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_PEER_NETWORKID=dev \
    -e CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft \
    -e CORE_PBFT_GENERAL_MODE=batch \
    -e CORE_PBFT_GENERAL_TIMEOUT_REQUEST=10s \
    hyperledger/fabric-peer:latest peer node start
```

vp1 ~ vp3

以 vp1 为例，假如 vp0 的地址为 10.0.0.1。

```
$ NAME=vp1
$ ROOT_NODE=10.0.0.1
$ docker run --name=${NAME} \
    --net="host" \
    --restart=unless-stopped \
    -it --rm \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -e CORE_PEER_ID=${NAME} \
    -e CORE_PBFT_GENERAL_N=4 \
    -e CORE_LOGGING_LEVEL=debug \
    -e CORE_PEER_ADDRESSAUTODETECT=true \
    -e CORE_PEER_NETWORKID=dev \
    -e CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft \
    -e CORE_PBFT_GENERAL_MODE=batch \
    -e CORE_PBFT_GENERAL_TIMEOUT_REQUEST=10s \
    -e CORE_PEER_DISCOVERY_ROOTNODE=${ROOT_NODE}:7051 \
    hyperledger/fabric-peer:latest peer node start
```

服务端口

Hyperledger 默认监听的服务端口包括：

- 7050: REST 服务端口，推荐 NVP 节点开放，0.6 之前版本中为 5000；
- 7051：peer gRPC 服务监听端口，0.6 之前版本中为 30303；
- 7052：peer CLI 端口，0.6 之前版本中为 30304；
- 7053：peer 事件服务端口，0.6 之前版本中为 31315；
- 7054：eCAP
- 7055：eCAA
- 7056：tCAP

- 7057：tCAA
- 7058：tlsCAP
- 7059：tlsCAA

使用 chaincode

下面演示 example02 chaincode，完成两方（如 a 和 b）之间进行价值的转移。

部署 chaincode

集群启动后，进入一个 VP 节点。以 pbft 模式为例，节点名称为 `pbft_vp0_1` 。

```
$ docker exec -it pbft_vp0_1 bash
```

部署 chaincode example02。

```
$ peer chaincode deploy -p github.com/hyperledger/fabric/examples/chaincode/go/chainco
de_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
03:08:44.740 [chaincodeCmd] chaincodeDeploy -> INFO 001 Deploy result: type:GOLANG cha
incodeID:<path:"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example0
2" name:"ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5
dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539" > ctorMsg:<args:"init" args:"a" a
rgs:"100" args:"b" args:"200" >
Deploy chaincode: ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb
9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
03:08:44.740 [main] main -> INFO 002 Exiting.....
```

返回 chaincode id 为 `ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539` ，后面将用这个 id 来标识这次交易。为了方便，把它记录到环境变量 CC_ID 中。

```
$ CC_ID="ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5
dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539"
```

部署成功后，系统中会自动生成几个 chaincode 容器，例如

```
CONTAINER ID        IMAGE
                                                                                COMMAND
            CREATED             STATUS              PORTS
      NAMES
e86c26bad76f        dev-vp1-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp1-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
597ebaf929a0        dev-vp2-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp2-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
8748a3b47312        dev-vp3-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp3-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
cf6e762f6a2e        dev-vp0-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05
eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539   "/opt/gopath/
bin/ee5b"   2 minutes ago       Up 2 minutes
      dev-vp0-ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070
549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
```

查询 chaincode

查询 a 手头的价值，为初始值 100。

```
$ peer chaincode query -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
03:22:31.420 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > >
Query Result: 100
03:22:31.420 [main] main -> INFO 002 Exiting.....
```

调用 chaincode

a 向 b 转账 10 元。

```
$ peer chaincode invoke -n ${CC_ID} -c '{"Function": "invoke", "Args": ["a", "b", "10"
]}'
03:22:57.345 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully invoked tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"invoke" args:"a" args:"b" args:"10" > > (fc298ffb-c763
-4ed0-9da2-072de2ab20b1)
03:22:57.345 [main] main -> INFO 002 Exiting.....
```

查询 a 手头的价值，为新的值 90。

```sh $ peer chaincode query -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}' 03:23:33.045 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Successfully queried transaction: chaincodeSpec: ctorMsg: > Query Result: 90 03:23:33.045 [main] main -> INFO 002 Exiting..... ···
```

权限管理

权限管理机制是 hyperledger fabric 项目的一大特色。下面给出使用权限管理的一个应用案例。

启动集群

首先下载相关镜像。

```
$ docker pull yeasy/hyperledger:latest
$ docker tag yeasy/hyperledger:latest hyperledger/fabric-baseimage:latest
$ docker pull yeasy/hyperledger-peer:latest
$ docker pull yeasy/hyperledger-membersrvc:latest
```

进入 hyperledger 项目，启动带成员管理的 PBFT 集群。

```
$ git clone https://github.com/yeasy/docker-compose-files
$ cd docker-compose-files/hyperledger/0.6/pbft
$ docker-compose -f 4-peers-with-membersrvc.yml up
```

用户登陆

当启用了权限管理后，首先需要登录，例如以内置账户 jim 账户登录。

登录 vp0，并执行登录命令。

```
$ docker exec -it pbft_vp0_1 bash
# peer network login jim
06:57:13.603 [networkCmd] networkLogin -> INFO 001 CLI client login...
06:57:13.603 [networkCmd] networkLogin -> INFO 002 Local data store for client loginTo
ken: /var/hyperledger/production/client/
Enter password for user 'jim': 6avZQLwcUe9b
06:57:25.022 [networkCmd] networkLogin -> INFO 003 Logging in user 'jim' on CLI interf
ace...
06:57:25.576 [networkCmd] networkLogin -> INFO 004 Storing login token for user 'jim'.
06:57:25.576 [networkCmd] networkLogin -> INFO 005 Login successful for user 'jim'.
06:57:25.576 [main] main -> INFO 006 Exiting.....
```

也可以用 REST 方式：

```
POST  HOST:7050/registrar
```

Request：

```
{
  "enrollId": "jim",
  "enrollSecret": "6avZQLwcUe9b"
}
```

Response：

```
{
    "OK": "User jim is already logged in."
}
```

chaincode 部署

登录之后，chaincode 的部署、调用等操作与之前类似，只是需要通过 -u 选项来指定用户名。

在 vp0 上执行命令：

```
# peer chaincode deploy -u jim -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -c '{"Function":"init", "Args": ["a","100", "b", "200"]}'
06:58:20.099 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is already logged in. Retrieving login token.
06:58:22.178 [chaincodeCmd] chaincodeDeploy -> INFO 002 Deploy result: type:GOLANG chaincodeID:<path:"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02" name:"ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539" > ctorMsg:<args:"init" args:"a" args:"100" args:"b" args:"200" >
Deploy chaincode: ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
06:58:22.178 [main] main -> INFO 003 Exiting.....
```

记录下返回的 chaincode ID。

```
# CC_ID=ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539
```

此时，查询账户值应当为初始值。

```
# peer chaincode query -u jim -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
07:28:39.925 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:28:40.281 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > secureContext:"jim" >
Query Result: 100
07:28:40.281 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "path":"github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example0
2"
    },
    "ctorMsg": {
        "function":"init",
        "args":["a", "1000", "b", "2000"]
    },
    "secureContext": "jim"
  },
  "id": 1
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "ee5b24a1f17c356dd5f6e37307922e39ddba12e5d2e203ed93401d7d05eb0dd194
fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab1812590cd0f78539"
    },
    "id": 1
}
```

chaincode 调用

在账户 a、b 之间进行转账 10 元的操作。

```
# peer chaincode invoke -u jim -n ${CC_ID} -c '{"Function": "invoke", "Args": ["a", "b
", "10"]}'
07:29:25.245 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:29:25.585 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully invoked tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"invoke" args:"a" args:"b" args:"10" > secureContext:"j
im" > (f8347e3b-7230-4561-9017-3946756a0bf4)
07:29:25.585 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"980d4bb7f69578592e5775a6da86d81a221887817d7164d3e9d4d4df1c981440abf9
a61417eaf8ad6f7fc79893da36de2cf4709131e9af39bca6ebc2e5a1cd9d"
      },
      "ctorMsg": {
         "function":"invoke",
         "args":["a", "b", "100"]
      },
    "secureContext": "jim"
  },
  "id": 3
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "66308740-a2c5-4a60-81f1-778dbed49cc3"
    },
    "id": 3
}
```

chaincode 查询

查询 a 账户的余额。

```
# peer chaincode query -u jim -n ${CC_ID} -c '{"Function": "query", "Args": ["a"]}'
07:29:55.844 [chaincodeCmd] getChaincodeSpecification -> INFO 001 Local user 'jim' is
already logged in. Retrieving login token.
07:29:56.198 [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 002 Successfully queried tr
ansaction: chaincodeSpec:<type:GOLANG chaincodeID:<name:"ee5b24a1f17c356dd5f6e37307922
e39ddba12e5d2e203ed93401d7d05eb0dd194fb9070549c5dc31eb63f4e654dbd5a1d86cbb30c48e3ab181
2590cd0f78539" > ctorMsg:<args:"query" args:"a" > secureContext:"jim" >
Query Result: 90
07:29:56.198 [main] main -> INFO 003 Exiting.....
```

也可以通过 REST 方式进行：

```
POST  HOST:7050/chaincode
```

Request：

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"980d4bb7f69578592e5775a6da86d81a221887817d7164d3e9d4d4df1c981440abf9
a61417eaf8ad6f7fc79893da36de2cf4709131e9af39bca6ebc2e5a1cd9d"
      },
      "ctorMsg": {
         "function":"query",
         "args":["a"]
      },
      "secureContext": "jim"
  },
  "id": 5
}
```

Response：

```
{
    "jsonrpc": "2.0",
    "result": {
        "status": "OK",
        "message": "900"
    },
    "id": 5
}
```

区块信息查询

URL：

```
GET  HOST:7050/chain/blocks/2
```

Response：

```
{
    "transactions": [
        {
            "type": 2,
            "chaincodeID": "EoABMjhiYjJiMjMxNjE3MWE3MDZiYjI4MTBlYzM1ZDA5NWY0MzA4NzdiZj
Q0M2YxMDYxZWYwZjYwYmJlNzUzZWQ0NDA3MDBhNTMxMmMxNjM5MGQzYjMwMTk5ZmU5NDY1YzNiNzVkNTk0NDM1
OGNhYWUwMWNhODFlZjI4MTI4YTFiZmI=",
            "payload": "Cp0BCAESgwESgAEyOGJiMmIyMzE2MTcxYTcwNmJiMjgxMGVjMzVkMDk1ZjQzMD
g3N2JmNDQzZjEwNjFlZjBmNjBiYmU3NTNlZDQ0MDcwMGE1MzEyYzE2MzkwZDNiMzAxOTlmZTk0NjVjM2I3NWQ1
OTQ0MzU4Y2FhZTAxY2E4MWVmMjgxMjhhMWJmYhoTCgZpbnZva2USAWESAWISAzEwMA==",
            "uuid": "2b3b6cf3-9887-4dd5-8f2e-3634ec9c719a",
            "timestamp": {
                "seconds": 1466577447,
                "nanos": 399637431
            },
            "nonce": "5AeA6S1odhPIDiGjFTFG8ttcihOoNNsh",
            "cert": "MIICPzCCAeSgAwIBAgIRAMndnS+Me0G6gs4J9/fb8HcwCgYIKoZIzj0EAwMwMTELM
AkGA1UEBhMCVVMxFDASBgNVBAoTC0h5cGVybGVkZ2VyMQwwCgYDVQQDEwN0Y2EwHhcNMTYwNjIyMDYzMzE4Whc
NMTYwOTIwMDYzMzE4WjAxMQswCQYDVQQGEwJVUzEUMBIGA1UEChMLSHlwZXJsZWRnZXIxDDAKBgNVBAMTA2ppb
TBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABDLd2W8PxzgB4A85Re2x44BApbOGqP05tnkygbXSctLiqi5HVfw
RAACS6znVA9+toni59Yy+XAH3w2offdjFW3mjgdwwgdkwDgYDVR0PAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwD
QYDVR0OBAYEBAECAwQwDwYDVR0jBAgwBoAEAQIDBDBNBgYqAwQFBgcBAf8EQAfASTE6bZ0P5mrEzTa5r1UyKFv
+dKezBiGU0V3l2iWzk9evlGMvaC2pwhEKfKDdKxs7YSMYe/7cLq/oF++GBVowSgYGKgMEBQYIBEBEO3TKXuORl
5Geuco8Gnn5TkoIl4+b96aPGDGvKbmDjMXR9vEBuUXTnsbDL53j7kC8/XQs1kZboC1ojLeUSN03MAoGCCqGSM4
9BAMDA0kAMEYCIQCZqyANMFcu1WiMe2So0pC7eRU95F0+qUXLAKZsPWv/YQIhALmNaglP7CoMOe2qxehucmffD
lu0BRLSYDHyV9xcxmkH",
            "signature": "MEYCIQDob3NqdrfwlSGhi+zz+Ypl7S9QQ07RIFr8nV92e8KDNgIhANIljz4t
RS8vwQk01hTemNQFJX2zMI6DhSUFZivbbtoR"
        }
    ],
    "stateHash": "7YUoVvYnMLHbLf47uTixLtkjF6xM9DuvgSWC92MbOUzk09xhcRBBLZqe5FvJElgZemEL
BOcuIFnubL0LiGH0yw==",
    "previousBlockHash": "On4BlpqCYNpugUKluqvOcbvkr3TAQxmlISLdd6qrONtIgmQ4iUDeWxAA9lUC
ceZfF8tke8A0Wy7m9tksNpKodw==",
    "consensusMetadata": "CAI=",
    "nonHashData": {
        "localLedgerCommitTimestamp": {
            "seconds": 1466577447,
            "nanos": 653618964
        },
        "transactionResults": [
            {
                "uuid": "2b3b6cf3-9887-4dd5-8f2e-3634ec9c719a"
            }
        ]
    }
}
```

Python 客户端

前面应用案例，都是直接通过 HTTP API 来跟 hyperledger 进行交互，操作比较麻烦。

还可以直接通过 hyperledger-py 客户端来进行更方便的操作。

安装

```
$ pip install hyperledger --upgrade
```

或直接源码安装

```
$ git clone https://github.com/yeasy/hyperledger-py.git
$ cd hyperledger-py
$ pip install -r requirements.txt
$ python setup.py install
```

使用

```
>>> from hyperledger.client import Client
>>> c = Client(base_url="http://127.0.0.1:7050")
>>> c.peer_list()
{u'peers': [{u'type': 1, u'ID': {u'name': u'vp1'}, u'address': u'172.17.0.2:30303'}, {u
'type': 1, u'ID': {u'name': u'vp2'}, u'address': u'172.17.0.3:30303'}]}
```

更多使用方法，可以参考 API 文档。

其它客户端

目前，HyperLedger Fabric 已经成立了 SDK 工作组。

目前在实现的客户端 SDK 包括：

- Python SDK
- Nodejs SDK

本章小结

区块链应用开发

简介

链码示例一：信息公证

简介

chaincode_example01.go 主要实现如下的功能：

- 初始化，以键值形式存放信息；
- 允许读取和修改键值。

代码中，首先初始化了 `hello_world` 的值，并根据请求中的参数创建修改查询链上 `key` 中的值，本质上实现了一个简单的可修改的键值数据库。

主要函数

- `read` ：读取key `args[0]` 的 value；
- `write` ：创建或修改 key `args[0]` 的 value；
- `init` ：初始化 key `hello_world` 的 value；
- `invoke` ：根据传递参数类型调用执行相应的 `init` 和 `write` 函数；
- `query` ：调用 `read` 函数查询 `args[0]` 的 value。

代码运行分析

`main` 函数作为程序的入口，调用 shim 包的 start 函数，启动 chaincode 引导程序的入口节点。如果报错，则返回。

```
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

当智能合约部署在区块链上，可以通过 rest api 进行交互。

三个主要的函数是 `init` ， `invoke` ， `query` 。在三个函数中，通过 `stub.PutState` 与 `stub.GetState` 存储访问 ledger 上的键值对。

通过 REST API 操作智能合约

假设以 jim 身份登录 pbft 集群，请求部署该 chaincode 的 json 请求格式为：

```
{
    "jsonrpc": "2.0",
    "method": "deploy",
    "params": {
        "type": 1,
        "chaincodeID": {
            "path": "https://github.com/ibm-blockchain/learn-chaincode/finished"
        },
        "ctorMsg": {
            "function": "init",
            "args": [
                "hi there"
            ]
        },
        "secureContext": "jim"
    },
    "id": 1
}
```

目前 path 仅支持 github 上的目录，ctorMsg 中为函数 `init` 的传参。

调用 invoke 函数的 json 格式为：

```
{
    "jsonrpc": "2.0",
    "method": "invoke",
    "params": {
        "type": 1,
        "chaincodeID": {
            "name": "4251b5512bad70bcd0947809b163bbc8398924b29d4a37554f2dc2b033617c19c
c0611365eb4322cf309b9a5a78a5dba8a5a09baa110ed2d8aeee186c6e94431"
        },
        "ctorMsg": {
            "function": "init",
            "args": [
                "swb"
            ]
        },
        "secureContext": "jim"
    },
    "id": 2
}
```

其中 name 字段为 `deploy` 后返回的 message 字段中的字符串。

`query` 的接口也是类似的。

链码示例二：交易资产

简介

chaincode_example02.go 主要实现如下的功能：

- 初始化 A、B 两个账户，并为两个账户赋初始资产值；
- 在 A、B 两个账户之间进行资产交易；
- 分别查询 A、B 两个账户上的余额，确认交易成功；
- 删除账户。

主要函数

- `init` ：初始化 A、B 两个账户；
- `invoke` ：实现 A、B 账户间的转账；
- `query` ：查询 A、B 账户上的余额；
- `delete` ：删除账户。

依赖的包

```
import (
    "errors"
    "fmt"
    "strconv"

    "github.com/hyperledger/fabric/core/chaincode/shim"
)
```

`strconv` 实现 int 与 string 类型之间的转换。

在invoke 函数中，存在：

```
X, err = strconv.Atoi(args[2])
    Aval = Aval - X
    Bval = Bval + X
```

当 `args[2]<0` 时，A 账户余额增加，否则 B 账户余额减少。

可扩展功能

实例中未包含新增账户并初始化的功能。开发者可以根据自己的业务模型进行添加。

数字货币发行与管理

简介

该智能合约实现一个简单的商业应用案例，即数字货币的发行与转账。在这之中一共分为三种角色：中央银行，商业银行，企业。其中中央银行可以发行一定数量的货币，企业之间可以进行相互的转账。主要实现如下的功能：

- 初始化中央银行及其发行的货币数量
- 新增商业银行，同时央行并向其发行一定数量的货币
- 新增企业
- 商业银行向企业转给一定数量的数字货币
- 企业之间进行相互的转账
- 查询企业、银行、交易信息

主要函数

- `init` ：初始化中央银行，并发行一定数量的货币；
- `invoke` ：调用合约内部的函数；
- `query` ：查询相关的信息；
- `createBank` ：新增商业银行，同时央行向其发行一定数量的货币；
- `createCompany` ：新增企业；
- `issueCoin` ：央行再次发行一定数量的货币（归于交易）；
- `issueCoinToBank` ：央行向商业银行转一定数量的数字货币（归于交易）；
- `issueCoinToCp` ：商业银行向企业转一定数量的数字货币（归于交易行为）；
- `transfer` ：企业之间进行相互转账（归于交易行为）；
- `getCompanys` ：获取所有的公司信息，如果企业个数大于10，先访问前10个；
- `getBanks` ：获取所有的商业银行信息，如果商业银行个数大于10，先访问前 10 个
- `getTransactions` ：获取所有的交易记录 如果交易个数大于10，先访问前 10 个；
- `getCompanyById` ：获取某家公司信息；
- `getBankById` ：获取某家银行信息；
- `getTransactionBy` ：获取某笔交易记录；
- `writeCenterBank` ：修改央行信息；
- `writeBank` ：修改商业银行信息；
- `writeCompany` ：修改企业信息；
- `writeTransaction` ：写入交易信息。

数据结构设计

- centerBank 中央银行
    - Name：名称
    - TotalNumber：发行货币总数额
    - RestNumber：账户余额
    - ID：ID固定为 0
- bank 商业银行
    - Name：名称
    - TotalNumber：收到货币总数额
    - RestNumber：账户余额
    - ID：银行 ID
- company 企业
    - Name：名称
    - Number：账户余额
    - ID：企业 ID
- transaction 交易内容
    - FromType：发送角色 //centerBank:0,Bank:1,Company:2
    - FromID：发送方 ID
    - ToType：接收方角色 //Bank:1,Company:2
    - ToID：接收方 ID
    - Time：交易时间
    - Number：交易数额
    - ID：交易 ID

接口设计

`init`

request 参数:

```
args[0] 银行名称
args[1] 初始化发布金额
```

response 参数:

```
{"Name":"XXX","TotalNumber":"0","RestNumber":"0","ID":"XX"}
```

`createBank`

request 参数:

```
args[0] 银行名称
```

response 参数:

```
{"Name":"XXX","TotalNumber":"0","RestNumber":"0","ID":"XX"}
```

`createCompany`

request 参数:

```
args[0] 公司名称
```

response 参数:

```
{"Name":"XXX","Number":"0","ID":"XX"}
```

`issueCoin`

request 参数:

```
args[0] 再次发行货币数额
```

response 参数:

```
{"FromType":"0","FromID":"0","ToType":"0","ToID":"0","Time":"XX","Number":"XX","ID":"X
X"}
```

`issueCoinToBank`

request 参数:

```
args[0] 商业银行ID
args[1] 转账数额
```

response 参数:

```
{"FromType":"0","FromID":"0","ToType":"1","ToID":"XX","Time":"XX","Number":"XX","ID":"
XX"}
```

`issueCoinToCp`

request 参数:

```
args[0] 商业银行ID
args[1] 企业ID
args[2] 转账数额
```

response 参数:

```
{"FromType":"1","FromID":"XX","ToType":"2","ToID":"XX","Time":"XX","Number":"XX","ID":
"XX"}
```

`transfer`

request 参数:

```
args[0] 转账用户ID
args[1] 被转账用户ID
args[2] 转账余额
```

response 参数:

```
{"FromType":"2","FromID":"XX","ToType":"2","ToID":"XX","Time":"XX","Number":"XX","ID":
"XX"}
```

`getBanks`

response 参数

```
[{"Name":"XXX","Number":"XX","ID":"XX"},{"Name":"XXX","Number":"XX","ID":"XX"},...]
```

`getCompanys`

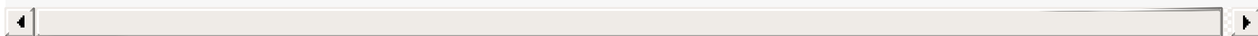
response 参数

```
[{"Name":"XXX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"},{"Name":"XXX","TotalNum
ber":"XX","RestNumber":"XX","ID":"XX"},...]
```

`getTransactions`

response 参数

```
[{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID":"XX"},{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID":"XX"},...]
```



`getCenterBank`

response 参数

```
[{"Name":"XX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"}]
```

`getBankById`

request 参数

```
args[0] 商业银行ID
```

response 参数

```
[{"Name":"XX","TotalNumber":"XX","RestNumber":"XX","ID":"XX"}]
```

`getCompanyById`

request 参数

```
args[0] 企业ID
```

response 参数

```
[{"Name":"XXX","Number":"XX","ID":"XX"}]
```

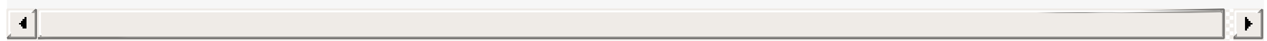
`getTransactionById`

request 参数

```
args[0] 交易ID
```

response 参数

```
{"FromType":"XX","FromID":"XX","ToType":"XX","ToID":"XX","Time":"XX","Number":"XX","ID"
:"XX"}
```



`writeCenterBank`

request 参数

```
CenterBank
```

response 参数

```
err  nil 为成功
```

`writeBank`

request 参数

```
Bank
```

response 参数

```
err  nil 为成功
```

`writeCompany`

request 参数

```
Company
```

response 参数

```
err  nil 为成功
```

`writeTransaction`

request 参数

```
Transaction
```

response 参数

``` err nil 为成功 ···

# 其它

查询时为了兼顾读速率，将一些信息备份存放在非区块链数据库上也是一个较好的选择。
```

学历认证

功能描述

该 智能合约 实现了一个简单的征信管理的案例。针对于学历认证领域，由于条约公开，在条约外无法随意篡改的特性，天然具备稳定性和中立性。

该智能合约中三种角色如下：

- 学校
- 个人
- 需要学历认证的机构或公司

学校可以根据相关信息在区块链上为某位个人授予学历，相关机构可以查询某人的学历信息，由于使用私钥签名，确保了信息的真实有效。 为了简单，尽量简化相关的业务，另未完成学业的学生因违纪或外出创业退学，学校可以修改其相应的学历信息。

账户私钥应该由安装在本地的客户端生成，本例中为了简便，使用模拟私钥和公钥。

数据结构设计

- 学校
  - 名称
  - 所在位置
  - 账号地址
  - 账号公钥
  - 账户私钥
  - 学校学生
- 个人
  - 姓名
  - 账号地址
  - 过往学历
- 学历信息
  - 学历信息编号
  - 就读学校
  - 就读年份
  - 完成就读年份
  - 就读状态 // 0:毕业 1：退学
- 修改记录（入学也相当于一种修改记录）
  - 编号
  - 学校账户地址（一般根据账户地址可以算出公钥地址，然后可以进行校验）

 - 学校签名
 - 个人账户地址
 - 个人公钥地址（个人不需要公钥地址）
 - 修改时间
 - 修改操作// 0:正常毕业 1：退学 2:入学

对学历操作信息所有的操作都归为记录。

function及各自实现的功能

- `init` 初始化函数
- `invoke` 调用合约内部的函数
- `updateDiploma` 由学校更新学生学历信息，并签名（返回记录信息）
- `enrollStudent` 学校招生（返回学校信息）
- `createSchool` 添加一名新学校
- `createStudent` 添加一名新学生
- `getStudentByAddress` 通过学生的账号地址访问学生的学历信息
- `getRecordById` 通过Id获取记录
- `getRecords` 获取全部记录（如果记录数大于 10，返回前 10 个）
- `getSchoolByAddress` 通过学校账号地址获取学校的信息
- `getBackgroundById` 通过学历 Id 获取所存储的学历信息
- `writeRecord` 写入记录
- `writeSchool` 写入新创建的学校
- `writeStudent` 写入新创建的学生

接口设计

`createSchool`

request参数:

```
args[0] 学校名称
args[1] 学校所在位置
```

response参数:

```
学校信息的字节数组，当创建一所新学校时，该学校学生账户地址列表为空
```

`createStudent`

request参数：

```
args[0] 学生的姓名
```

response参数：

```
学生信息的字节数组表示，刚创建过往学历信息列表为空
```

`updateDiploma`

request参数

```
args[0] 学校账户地址
args[1] 学校签名
args[2] 待修改学生的账户地址
args[3] //对该学生的学历进行怎样的修改，0：正常毕业  1：退学
```

response参数

```
返回修改记录的字节数组表示
```

`enrollStudent`

request参数:

```
args[0] 学校账户地址
args[1] 学校签名
args[2] 学生账户地址
```

response参数

```
返回修改记录的字节数组表示
```

`getStudentByAddress`

request参数

```
args[0] address
```

response参数

```
学生信息的字节数组表示
```

`getRecordById`

request参数

```
args[0] 修改记录的ID
```

response参数

```
修改记录的字节数组表示
```

`getRecords`

response参数

```
获取修改记录数组（如果个数大于10，返回前10个）
```

`getSchoolByAddress`

request参数

```
args[0] address
```

response参数

```
学校信息的字节数组表示
```

`getBackgroundById`

request参数

```
args[0] ID
```

response参数

```
学历信息的字节数组表示
```

测试

社区能源共享

功能描述

本 合约 以纽约实验性的能源微电网为例，作为一个简单的案例进行实现。

> “在总统大道的一边，五户家庭通过太阳能板发电；在街道的另一边的五户家庭可以购买对面家庭不需要的电力。而连接这项交易的就是区块链网络，几乎不需要人员参与就可以管理记录交易。”但是这个想法是非常有潜力的，能够代表未来社区管理能源系统。”

布鲁克林微电网开发商 LO3 创始人 Lawrence Orsini 说：

> “我们正在这条街道上建立一个可再生电力市场，来测试人们对于购买彼此手中的电力是否感兴趣。如果你在很远的地方生产能源，运输途中会有很多损耗，你也得不到这电力价值。但是如果你就在街对面，你就能高效的利用能源。”

在某一块区域内存在一个能源微电网，每一户家庭可能为生产者也可能为消费者。部分家庭拥有太阳能电池板，太阳能电池板的剩余电量为可以售出的电力的值，为了简化，单位为1.需要电力的家庭可以向有足够余额的电力的家庭购买电力。

账户私钥应该由安装在本地的客户端生成，本例中为了简便，使用模拟私钥和公钥。每位用户的私钥为guid+“1”，公钥为guid+“2”。签名方式简化为私钥+"1"

数据结构设计

在该智能合约中暂时只有一种角色，为每一户家庭用户。

- 家庭用户
  - 账户地址
  - 剩余能量 //部分家庭没有太阳能电池板，值为0
  - 账户余额（电子货币）
  - 编号
  - 状态 //0：不可购买， 1：可以购买
  - 账户公钥
  - 账户私钥
- 交易(一笔交易必须同时具有卖方和买方的公钥签名，方能承认这笔交易。公钥签名生成规则，公钥+待创建交易的ID号，在本交易类型中，只要买家有足够的货币，卖家自动会对交易进行签名)
  - 购买方地址
  - 销售方地址
  - 电量销售量
  - 电量交易金额

  - 编号
  - 交易时间

function及各自实现的功能

- `init` 初始化操作
- `invoke` 调用合约内部的函数
- `query` 查询相关的信息
- `createUser` 创建新用户，并加入到能源微网中 invoke
- `buyByAddress` 向某一位用户购买一定量的电力 invoke
- `getTransactionById` 通过id获取交易内容 query
- `getTransactions` 获取交易（如果交易数大于10，获取前10个） query
- `getHomes` 获取用户（如果用户数大于10，获取前10个） query
- `getHomeByAddress` 通过地址获取用户 query
- `changeStatus` 某一位用户修改自身的状态 invoke
- `writeUser` 将新用户写入到键值对中
- `writeTransaction` 记录交易

接口设计

`createUser`

request参数:

```
args[0] 剩余能量值
args[1] 剩余金额
```

response参数:

```
新建家庭用户的json表示
```

`buyByAddress`

request参数:

```
args[0] 卖家的账户地址
args[1] 买家签名
args[2] 买家的账户地址
args[3] 想要购买的电量数值
```

response参数:

```
购买成功的话返回该transaction的json串。
购买失败返回error
```

`getTransactionById`

request参数:

```
args[0] 交易编号
```

response参数:

```
查询结果的transaction 交易表示
```

`getTransactions`

request参数:

```
none
```

response参数:

```
获取所有的交易列表（如果交易大于10，则返回前10个）
```

`getHomeByAddress`

request参数

```
args[0] address
```

response参数

```
用户信息的json表示
```

`getHomes`

response参数

```
获取所有的用户列表（如果用户个数大于10，则返回前10个）
```

`changeStatus`

request参数:

```
args[0] 账户地址
args[1]  账户签名
args[2] 对自己的账户进行的操作，0：设置为不可购买   1：设置状态为可购买
```

response参数:

```
修改后的用户信息json表示
```

测试

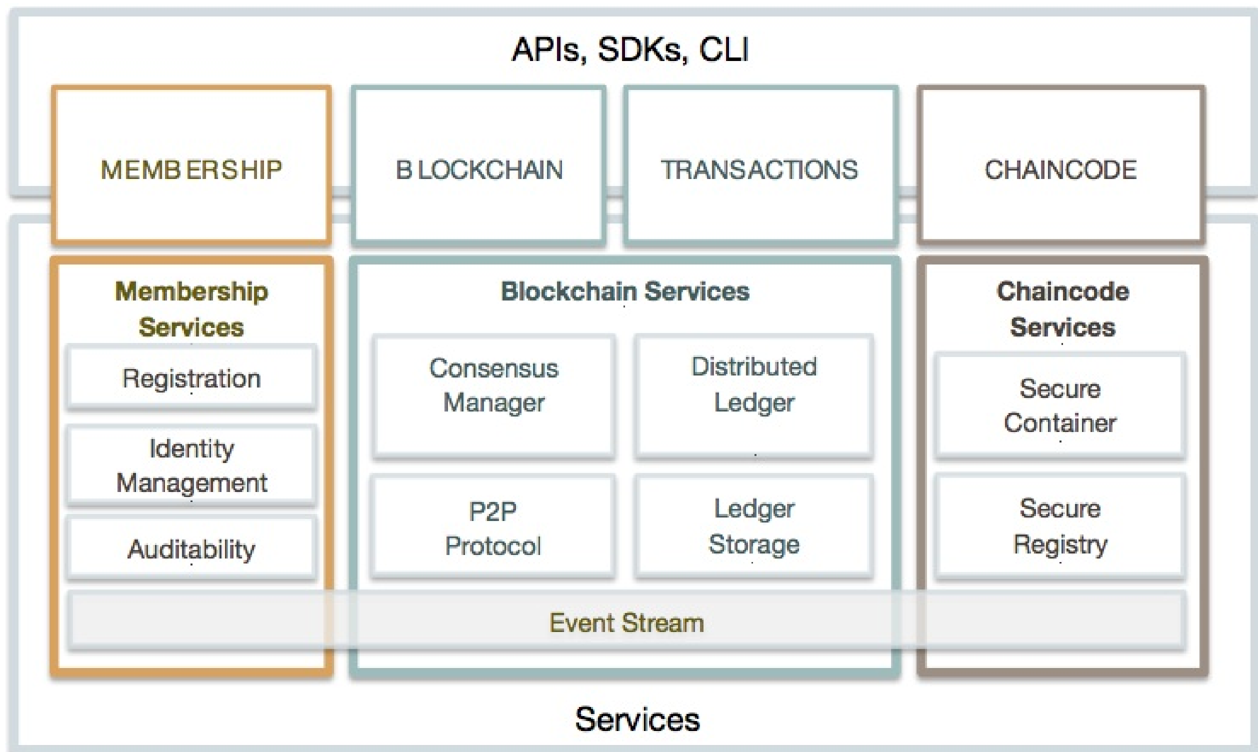
本章小结

Fabric 架构与设计

简介

架构设计

整个功能架构如下图所示。



包括三大组件：区块链服务（Blockchain）、链码服务（Chaincode）、成员权限管理（Membership）。

概念术语

- Auditability（审计性）：在一定权限和许可下，可以对链上的交易进行审计和检查。
- Block（区块）：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。
- Blockchain（区块链）：由多个区块链接而成的链表结构，除了首个区块，每个区块都包括前继区块内容的 hash 值。
- Certificate Authority（CA）：负责身份权限管理，又叫 Member Service 或 Identity Service。
- Chaincode（链上代码或链码）：区块链上的应用代码，扩展自“智能合约”概念，支持 golang、nodejs 等，运行在隔离的容器环境中。
- Committer（提交节点）：1.0 架构中一种 peer 节点角色，负责对 orderer 排序后的交易进行检查，选择合法的交易执行并写入存储。
- Confidentiality（保密）：只有交易相关方可以看到交易内容，其它人未经授权则无法看到。
- Endorser（背书节点）：1.0 架构中一种 peer 节点角色，负责检验某个交易是否合法，是否愿意为之背书、签名。

* Enrollment Certificate Authority（ECA，注册 CA）：负责成员身份相关证书管理的 CA。
* Ledger（账本）：包括区块链结构（带有所有的可验证交易信息，但只有最终成功的交易会改变世界观）和当前的世界观（world state）。Ledger 仅存在于 Peer 节点。
* MSP（Member Service Provider，成员服务提供者）：成员服务的抽象访问接口，实现对不同成员服务的可拔插支持。
* Non-validating Peer（非验证节点）：不参与账本维护，仅作为交易代理响应客户端的 REST 请求，并对交易进行一些基本的有效性检查，之后转发给验证节点。
* Orderer（排序节点）：1.0 架构中的共识服务角色，负责排序看到的交易，提供全局确认的顺序。
* Permissioned Ledger（带权限的账本）：网络中所有节点必须是经过许可的，非许可过的节点则无法加入网络。
* Privacy（隐私保护）：交易员可以隐藏交易的身份，其它成员在无特殊权限的情况下，只能对交易进行验证，而无法获知身份信息。
* Transaction（交易）：执行账本上的某个函数调用。具体函数在 chaincode 中实现。
* Transactor（交易者）：发起交易调用的客户端。
* Transaction Certificate Authority（TCA，交易 CA）：负责维护交易相关证书管理的 CA。
* Validating Peer（验证节点）：维护账本的核心节点，参与一致性维护、对交易的验证和执行。
* World State（世界观）：是一个键值数据库，chaincode 用它来存储交易相关的状态。

区块链服务

区块链服务提供一个分布式账本平台。一般地，多个交易被打包进区块中，多个区块构成一条区块链。区块链代表的是账本状态机发生变更的历史过程。

交易

交易意味着围绕着某个链码进行操作。

交易可以改变世界状态。

交易中包括的内容主要有：

* 交易类型：目前包括 Deploy、Invoke、Query、Terminate 四种；
* uuid：代表交易的唯一编号；
* 链码编号 chaincodeID：交易针对的链码；
* 负载内容的 hash 值：Deploy 或 Invoke 时候可以指定负载内容；
* 交易的保密等级 ConfidentialityLevel；
* 交易相关的 metadata 信息；
* 临时生成值 nonce：跟安全机制相关；

- 交易者的证书信息 cert；
- 签名信息 signature；
- metadata 信息；
- 时间戳 timestamp。

交易的数据结构（Protobuf 格式）定义为

```
message Transaction {
    enum Type {
        UNDEFINED = 0;
        // deploy a chaincode to the network and call `Init` function
        CHAINCODE_DEPLOY = 1;
        // call a chaincode `Invoke` function as a transaction
        CHAINCODE_INVOKE = 2;
        // call a chaincode `query` function
        CHAINCODE_QUERY = 3;
        // terminate a chaincode; not implemented yet
        CHAINCODE_TERMINATE = 4;
    }
    Type type = 1;
    //store ChaincodeID as bytes so its encrypted value can be stored
    bytes chaincodeID = 2;
    bytes payload = 3;
    bytes metadata = 4;
    string uuid = 5;
    google.protobuf.Timestamp timestamp = 6;

    ConfidentialityLevel confidentialityLevel = 7;
    string confidentialityProtocolVersion = 8;
    bytes nonce = 9;

    bytes toValidators = 10;
    bytes cert = 11;
    bytes signature = 12;
}
```

在 1.0 架构中，一个 transaction 包括如下信息：

[ledger] [channel], **proposal:**[chaincode,] **endorsement:**[proposal hash, simulation result, signature]

- endorsements: proposal hash, simulation result, signature
- function-spec: function name, arguments
- proposal: [channel,] chaincode,

区块

区块打包交易，确认交易后的世界状态。

一个区块中包括的内容主要有：

- 版本号 version：协议的版本信息；
- 时间戳 timestamp：由区块提议者设定；
- 交易信息的默克尔树的根 hash 值：由区块所包括的交易构成；
- 世界观的默克尔树的根 hash 值：由交易发生后整个世界的状态值构成；
- 前一个区块的 hash 值：构成链所必须；
- 共识相关的元数据：可选值；
- 非 hash 数据：不参与 hash 过程，各个 peer 上的值可能不同，例如本地提交时间、交易处理的返回值等；

注意具体的交易信息并不存放在区块中。

区块的数据结构（Protobuf 格式）定义为

```
message Block {
    uint32 version = 1;
    google.protobuf.Timestamp timestamp = 2;
    repeated Transaction transactions = 3;
    bytes stateHash = 4;
    bytes previousBlockHash = 5;
    bytes consensusMetadata = 6;
    NonHashData nonHashData = 7;
}
```

一个真实的区块内容示例：

```
{
    "nonHashData": {
        "localLedgerCommitTimestamp": {
            "nanos": 975295157,
                "seconds": 1466057539
        },
            "transactionResults": [
            {
                "uuid": "7be1529ee16969baf9f3156247a0ee8e7eee99a6a0a816776acff65e6e1de
f71249f4cb1cad5e0f0b60b25dd2a6975efb282741c0e1ecc53fa8c10a9aaa31137"
            }
            ]
    },
        "previousBlockHash": "RrndKwuojRMjOz/rdD7rJD/NUupiuBuCtQwnZG7Vdi/XXcTd2MDyAMsF
AZ1ntZL2/IIcSUeatIZAKS6ss7fEvg==",
        "stateHash": "TiIwROg48Z4xXFFIPEunNpavMxnvmZKg+yFxKK3VBY0zqiK3L0QQ5ILIV85iy7U+
EiVhwEbkBb1Kb7w1ddqU5g==",
        "transactions": [
        {
            "chaincodeID": "CkdnaXRodWIuY29tL2h5cGVybGVkZ2VyL2ZhYnJpYy9leGFtcGxlcy9jaG
FpbmNvZGUvZ28vY2hhaW5jb2RlX2V4YW1wbGUwMhKAATdiZTE1MjllZTE2OTY5YmFmOWYzMTU2MjQ3YTBlZThl
N2VlZTk5YTZhMGE4MTY3NzZhY2ZmNjVlNmUxZGVmNzEyNDlmNGNiMWNhZDVlMGYwYjYwYjI1ZGQyYTY5NzVlZm
IyODI3NDFjMGUxZWNjNTNmYThjMTBhOWFhYTMxMTM3",
            "payload": "Cu0BCAESzAEKR2dpdGh1Yi5jb20vaHlwZXJsZWRnZXIvZmFicmljL2V4YW1wbG
VzL2NoYWluY29kZS9nby9jaGFpbmNvZGVfZXhhbXBsZTAyEoABN2JlMTUyOWVlMTY5NjliYWY5ZjMxNTYyNDdh
MGVlOGU3ZWVlOTlhNmEwYTgxNjc3NmFjZmY2NWU2ZTFkZWY3MTI0OWY0Y2IxY2FkNWUwZjBiNjBiMjVkZDJhNj
k3NWVmYjI4Mjc0MWMwZTFlY2M1M2ZhOGMxMGE5YWFhMzExMzcaGgoEaW5pdBIBYRIFMTAwMDASAWISBTIwMDAw"
,
            "timestamp": {
                "nanos": 298275779,
                "seconds": 1466057529
            },
            "type": 1,
            "uuid": "7be1529ee16969baf9f3156247a0ee8e7eee99a6a0a816776acff65e6e1def712
49f4cb1cad5e0f0b60b25dd2a6975efb282741c0e1ecc53fa8c10a9aaa31137"
        }
    ]
}
```


世界观

世界观用于存放链码执行过程中涉及到的状态变量，是一个键值数据库。典型的元素为 `[chaincodeID, ckey]: value` 结构。

为了方便计算变更后的 hash 值，一般采用默克尔树数据结构进行存储。树的结构由两个参数（ `numBuckets` 和 `maxGroupingAtEachLevel` ）来进行初始配置，并由 `hashFunction` 配置决定存放键值到叶子节点的方式。显然，各个节点必须保持相同的配置，并且启动后一般不建议变动。

- `numBuckets` ：叶子节点的个数，每个叶子节点是一个桶（bucket），所有的键值被

`hashFunction` 散列分散到各个桶，决定树的宽度；

- `maxGroupingAtEachLevel` ：决定每个节点由多少个子节点的 hash 值构成，决定树的深度。

其中，桶的内容由它所保存到键值先按照 chaincodeID 聚合，再按照升序方式组成。

一般地，假设某桶中包括 个 chaincodeID，对于 ，假设其包括 个键值对，则聚合 G_i 内容可以计算为：

该桶的内容则为

注：这里的 `+` 代表字符串拼接，并非数学运算。

链码服务

链码包含所有的处理逻辑，并对外提供接口，外部通过调用链码接口来改变世界观。

接口和操作

链码需要实现 Chaincode 接口，以被 VP 节点调用。

```
type Chaincode interface { Init(stub *ChaincodeStub, function string, args []string) ([
]byte, error) Invoke(stub *ChaincodeStub, function string, args []string) ([]byte, err
or) Query(stub *ChaincodeStub, function string, args []string) ([]byte, error)}
```

链码目前支持的交易类型包括：部署（Deploy）、调用（Invoke）和查询（Query）。

- 部署：VP 节点利用链码创建沙盒，沙盒启动后，处理 protobuf 协议的 shim 层一次性发送包含 ChaincodeID 信息的 REGISTER 消息给 VP 节点，进行注册，注册完成后，VP 节点通过 gRPC 传递参数并调用链码 Init 函数完成初始化；
- 调用：VP 节点发送 TRANSACTION 消息给链码沙盒的 shim 层，shim 层用传过来的参数调用链码的 Invoke 函数完成调用；
- 查询：VP 节点发送 QUERY 消息给链码沙盒的 shim 层，shim 层用传过来的参数调用链码的 Query 函数完成查询。

不同链码之间可能互相调用和查询。

容器

在实现上，链码需要运行在隔离的容器中，超级账本采用了 Docker 作为默认容器。

对容器的操作支持三种方法：build、start、stop，对应的接口为 VM。

```
type VM interface {
  build(ctxt context.Context, id string, args []string, env []string, attachstdin bool,
 attachstdout bool, reader io.Reader) error
  start(ctxt context.Context, id string, args []string, env []string, attachstdin bool,
 attachstdout bool) error
  stop(ctxt context.Context, id string, timeout uint, dontkill bool, dontremove bool)
error
}
```

链码部署成功后，会创建连接到部署它的 VP 节点的 gRPC 通道，以接受后续 Invoke 或 Query 指令。

gRPC 消息

VP 节点和容器之间通过 gRPC 消息来交互。消息基本结构为

```
message ChaincodeMessage {

 enum Type { UNDEFINED = 0; REGISTER = 1; REGISTERED = 2; INIT = 3; READY = 4; TRANSAC
TION = 5; COMPLETED = 6; ERROR = 7; GET_STATE = 8; PUT_STATE = 9; DEL_STATE = 10; INVO
KE_CHAINCODE = 11; INVOKE_QUERY = 12; RESPONSE = 13; QUERY = 14; QUERY_COMPLETED = 15;
 QUERY_ERROR = 16; RANGE_QUERY_STATE = 17; }

 Type type = 1; google.protobuf.Timestamp timestamp = 2; bytes payload = 3; string uui
d = 4;}
```

当发生链码部署时，容器启动后发送 `REGISTER` 消息到 VP 节点。如果成功，VP 节点返回 `REGISTERED` 消息，并发送 `INIT` 消息到容器，调用链码中的 Init 方法。

当发生链码调用时，VP 节点发送 `TRANSACTION` 消息到容器，调用其 Invoke 方法。如果成功，容器会返回 `RESPONSE` 消息。

类似的，当发生链码查询时，VP 节点发送 `QUERY` 消息到容器，调用其 Query 方法。如果成功，容器会返回 `RESPONSE` 消息。

成员权限管理

通过基于 PKI 的成员权限管理，平台可以对接入的节点和客户端的能力进行限制。

证书有三种，Enrollment，Transaction，以及确保安全通信的 TLS 证书。

- 注册证书 ECert：颁发给提供了注册凭证的用户或节点，一般长期有效；
- 交易证书 TCert：颁发给用户，控制每个交易的权限，一般针对某个交易，短期有效。
- 通信证书 TLSCert：控制对网络的访问，并且防止窃听。



新的架构设计

目前，VP 节点执行了所有的操作，包括接收交易，进行交易验证，进行一致性达成，进行账本维护等。这些功能的耦合导致节点性能很难进行扩展。

新的思路就是对这些功能进行解耦，让每个功能都相对单一，容易进行扩展。社区内已经有了一些讨论。

Fabric 1.0 的设计采用了适当的解耦，根据功能将节点角色解耦开，让不同节点处理不同类型的工作负载。

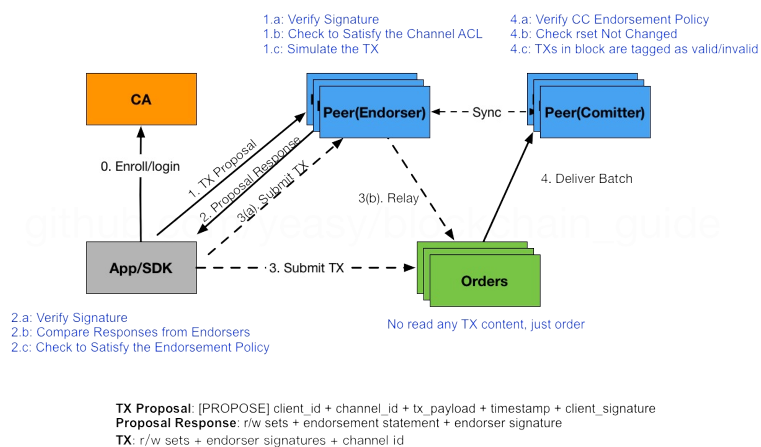


图 *1.14.2.1* - 示例工作过程

- 客户端：客户端应用使用 SDK 来跟 Fabric 打交道，构造合法的交易提案提交给 endorser；收集到足够多 endorser 支持后可以构造合法的交易请求，发给 orderer 或代理节点。
- Endorser peer：负责对来自客户端的交易进行合法性和 ACL 权限检查（模拟交易），通过则签名并返回结果给客户端。
- Committer peer：负责维护账本，将达成一致顺序的批量交易结果进行状态检查，生成区块，执行合法的交易，并写入账本，同一个物理节点可以同时担任 endorser 和 committer 的 角色。
- Orderer：仅负责排序，给交易们一个全局的排序，一般不需要跟账本和交易内容打交道。
- CA：负责所有证书的维护，遵循 PKI。

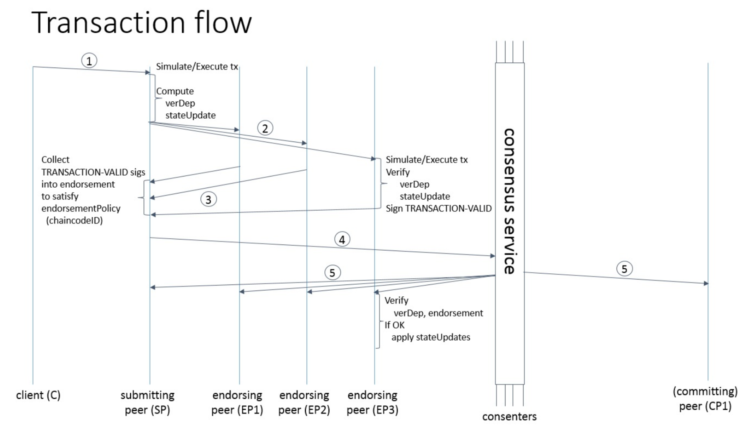


图 *1.14.2.2* - 示例交易过程

消息协议

节点之间通过消息来进行交互，所有消息都由下面的数据结构来实现。

```
message Message {
    enum Type {
        UNDEFINED = 0;

        DISC_HELLO = 1;
        DISC_DISCONNECT = 2;
        DISC_GET_PEERS = 3;
        DISC_PEERS = 4;
        DISC_NEWMSG = 5;

        CHAIN_STATUS = 6;
        CHAIN_TRANSACTION = 7;
        CHAIN_GET_TRANSACTIONS = 8;
        CHAIN_QUERY = 9;

        SYNC_GET_BLOCKS = 11;
        SYNC_BLOCKS = 12;
        SYNC_BLOCK_ADDED = 13;

        SYNC_STATE_GET_SNAPSHOT = 14;
        SYNC_STATE_SNAPSHOT = 15;
        SYNC_STATE_GET_DELTAS = 16;
        SYNC_STATE_DELTAS = 17;

        RESPONSE = 20;
        CONSENSUS = 21;
    }
    Type type = 1;
    bytes payload = 2;
    google.protobuf.Timestamp timestamp = 3;
}
```

消息分为四大类：Discovery（探测）、Transaction（交易）、Synchronization（同步）、Consensus（一致性）。

不同消息类型，对应到 payload 中数据不同，分为对应的子类消息结构。

Discovery

包括 DISC_HELLO、DISC_GET_PEERS、DISC_PEERS。

DISC_HELLO 消息结构如下。

```
message HelloMessage { PeerEndpoint peerEndpoint = 1; uint64 blockNumber = 2;}message PeerEndpoint { PeerID ID = 1; string address = 2; enum Type { UNDEFINED = 0; VALIDATOR = 1; NON_VALIDATOR = 2; } Type type = 3; bytes pkiID = 4;}

message PeerID { string name = 1;}
```

节点新加入网络时，会向 `CORE_PEER_DISCOVERY_ROOTNODE` 发送 `DISC_HELLO` 消息，汇报本节点的信息（id、地址、block 数、类型等），开始探测过程。

探测后发现 block 数落后对方，则会触发同步过程。

之后，定期发送 `DISC_GET_PEERS` 消息，获取新加入的节点信息。收到 `DISC_GET_PEERS` 消息的节点会通过 `DISC_PEERS` 消息返回自己知道的节点列表。

Transaction

包括 Deploy、Invoke、Query。消息结构如下：

```
message Transaction { enum Type { UNDEFINED = 0; CHAINCODE_DEPLOY = 1; CHAINCODE_INVOKE = 2; CHAINCODE_QUERY = 3; CHAINCODE_TERMINATE = 4; } Type type = 1; string uuid = 5; bytes chaincodeID = 2; bytes payloadHash = 3;

 ConfidentialityLevel confidentialityLevel = 7; bytes nonce = 8; bytes cert = 9; bytes signature = 10;

 bytes metadata = 4; google.protobuf.Timestamp timestamp = 6;}

message TransactionPayload { bytes payload = 1;}

enum ConfidentialityLevel { PUBLIC = 0; CONFIDENTIAL = 1;}
```

Synchronization

当节点发现自己 block 落后网络中最新状态，则可以通过发送如下消息（由 consensus 策略决定）来获取对应的返回。

- SYNC_GET_BLOCKS（对应 SYNC_BLOCKS）：获取给定范围内的 block 数据；
- SYNC_STATE_GET_SNAPSHOT（对应 SYNC_STATE_SNAPSHOT）：获取最新的世界观快照；
- SYNC_STATE_GET_DELTAS（对应 SYNC_STATE_DELTAS）：获取某个给定范围内的 block 对应的状态变更。

Consensus

consensus 组件收到 `CHAIN_TRANSACTION` 类消息后，将其转换为 `CONENSUS` 消息，然后向所有的 VP 节点广播。

收到 `CONSENSUS` 消息的节点会按照预定的 consensus 算法进行处理。

本章小结

区块链服务平台设计

规模是困难之源。

信息产业过去的十年，是云计算的十年。云计算技术为传统信息行业带来了前所未有的便捷。用户无需在意底层实现细节，通过简单的操作，即可获得可用的计算资源，节约大量运维管理的时间成本。

区块链平台作为分布式基础设施，其部署和维护过程需要多方面的技能，这对很多应用开发者来说都是不小的挑战。为了解决这些问题，区块链即服务（Blockchain as a Service, BaaS）平台应运而生。BaaS 可以利用云服务基础设施的部署和管理优势，为开发者提供创建、使用，甚至安全监控区块链平台的快捷服务。目前，业界已有一些区块链前沿技术团队率先开发并上线了区块链服务平台。

本章将首先介绍 BaaS 的概念，之后分别介绍业界领先的 IBM Bluemix 和微软 Azure 云上所提供的区块链服务。最后，还介绍了超级账本的区块链管理平台 —— Cello 项目，以及如何使用它快速搭建一套可以个性化的区块链服务平台。

简介

区块链即服务（Blockchain as a Service，BaaS），是部署在云计算基础设施之上，对外提供区块链网络的生命周期管理和运行时服务管理等功能的一套工具。

构建一套分布式的区块链方案绝非易事，既需要硬件基础设施的投入，也需要全方位的开发和运营管理（DevOps）。BaaS 作为一套工具，可以帮助开发者快速生成必要的区块链环境，进而验证所开发的上层应用。

除了区块链平台本身，一套完整的解决方案实际上还可以包括设备接入、访问控制、服务监控等管理功能。这些功能，让 BaaS 平台可以为开发者提供更强大的服务支持。

从 2016 年起，业界已有一些前沿技术团队发布了 BaaS 平台，除了商业的方案如 IBM Bluemix 和微软 Azure 云之外，超级账本开源项目也发起了 Cello 项目，以提供一套实现区块链平台运营管理功能的开源框架。

参考架构

下图给出了区块链即服务功能的参考架构，自上而下分为多层结构。最上层面向应用开发者和平台管理员提供不同的操作能力；核心层负责完成包括资源编排、系统监控、数据分析和权限管理等重要功能；下层可以通过多种类型的驱动和代理组件来访问和管理多种物理资源。

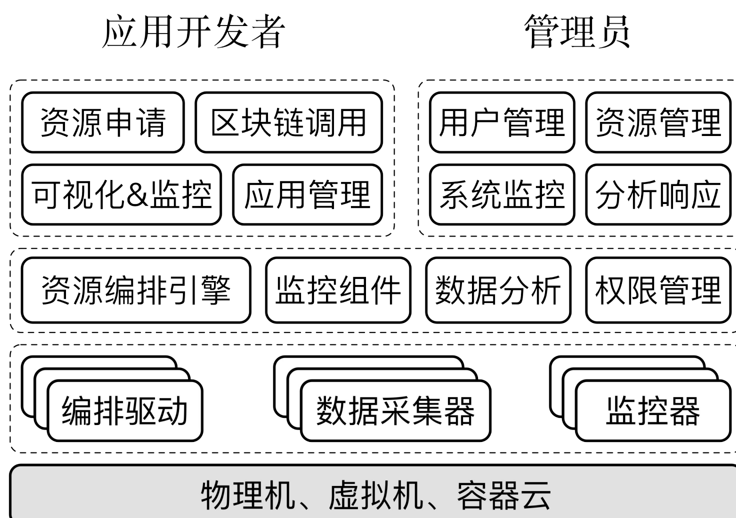


图 *1.15.1.1* - 区块链服务参考架构

典型地，BaaS 平台所提供的业务能力通常包括：

- 用户按需申请区块链网络，以及所需的计算、存储与网络连接资源；
- 用户对申请到的区块链进行生命周期管理，甚至支持灵活、弹性的区块链配置；
- 通过提供接口，让用户自由访问所申请到的区块链网络并进行调用；
- 提供直观的区块链可视化监控与操作界面，将区块链应用与底层平台无缝对接；
- 提供简单易用的智能合约开发与测试环境，方便用户对应用代码进行管理；
- 为管理员提供用户管理和资源管理操作；
- 为管理员提供对系统各项健康状态的实时监控；
- 提供对平台内各项资源和应用层的数据分析和响应能力。

考量指标

对于 BaaS 服务提供方，搭建这样一套功能完善、性能稳定的 BaaS 平台存在诸多挑战。可以从如下几个角度进行考量设计。

- 性能保障：包括区块链和应用的响应速度，监控实时性等；
- 可扩展性：支持大规模场景下部署和管理的能力，可以快速进行扩展；
- 资源调度：对于非均匀的资源请求类型可以智能的予以平缓化处理，合理分配系统资源；
- 安全性：注意平衡用户操作区块链的自由度与平台自身的安全可控；
- 可感知性：深度感知数据行为，如可以准确实时评估区块链的运行状况，给用户启发；
- 底层资源普适性：底层应当支持多种混合计算架构，容易导入物理资源。

此外，对于面向开发者的 BaaS 服务，创建的区块链环境应当尽量贴近实际应用场景，让用户可以将经过检验的区块链模型很容易地迁移到生产环境。甚至可以直接联动支持第三方发布平台，直接将经过验证的应用推向发布环境。

IBM Bluemix 云区块链服务

Bluemix 是 IBM 推出的开放的 PaaS 云平台，包含大量平台和软件服务，旨在帮助开发者实现一站式地应用开发与部署管理。

2016 年，Bluemix 面向开发者推出了基于超级账本 Fabric 的区块链服务，供全球的区块链爱好者使用。用户可以通过访问 https://console.ng.bluemix.net/catalog/services/blockchain 使用该服务。

服务介绍

Bluemix 为用户提供了在云上灵活管理超级账本 Fabric 区块链网络的能力，让开发者专注于快速创建、操作和监控区块链网络，而无需过多考虑底层硬件资源。同时，Bluemix 云平台本身也提供了安全、隐私性方面的保障，并对相关资源进行了性能优化。

Bluemix 目前提供了几种不同类型的区块链网络部署方案，包括免费的基础套餐到收费的高性能方案等。不同方案针对开发者的不同需求，在运行环境、占用资源、配置方式上都有所区别。

对于超级账本 Fabric 网络试用者，可选择免费的基础套餐，获得一个包含各类型 Peer 节点和 CA 的完整区块链试用网络，用户可以自行尝试部署链码并实时观察账本状态的变化。

使用服务

Bluemix 云平台提供的仪表盘（Dashboard）提供了十分直观的管理方式，用户可以通过 Web 界面来获取和访问区块链资源。

如下图所示，用户创建网络后，可以进入 Dashboard 看到属于自己的区块链网络，同时观察各节点的状态，以及与身份认证相关的服务凭证。

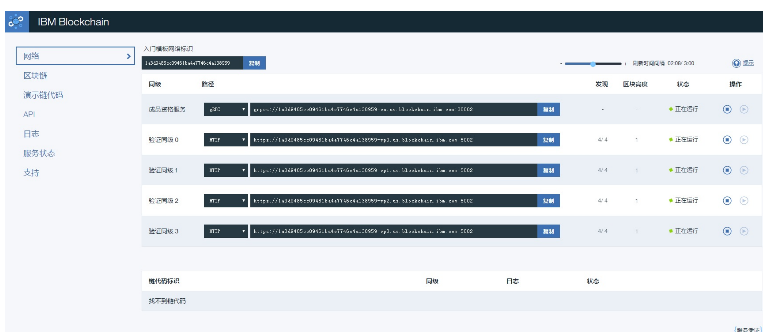


图 *1.15.2.1 - Bluemix* 区块链服务仪表盘

对于已经申请到的区块链网络，用户可以通过 Dashboard 对其部署并调用链码，并实时查看响应结果。例如，下图中展示了部署自带的 example02 链码。



图 *1.15.2.2 -* 通过 *Dashboard* 操作链码

对链码的操作会发送交易，进而生成新的区块。可通过 Dashboard 观察与区块链状态、区块内容相关的信息。例如，下图中区块链生成了 4 个区块，并执行了 1 次部署和 2 次调用。

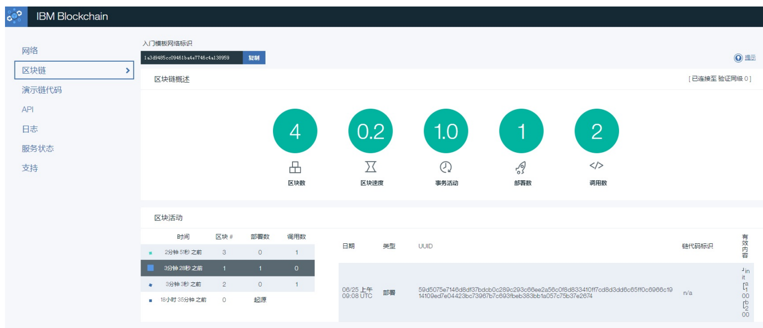


图 *1.15.2.3 -* 通过 *Dashboard* 观察区块链

平台同时会收集各节点的日志信息，监控和记录服务的运行状态。用户同样可以在 Dashboard 中实时查看。如下图所示，显示了服务和网络的正常运行时间等。

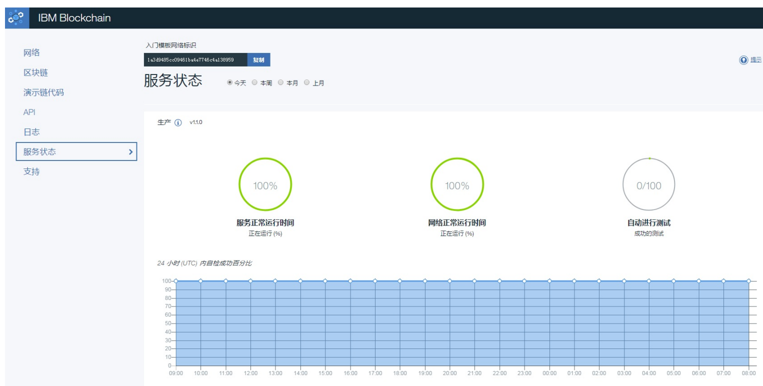


图 *1.15.2.4 -* 通过 *Dashboard* 获取服务状态

同时，Bluemix 云平台会将与区块链网络交互所需的 gRPC 或 HTTP 接口地址开放给用户，供用户通过 SDK 等进行远程操作，实现更多跟区块链、链码和应用相关的丰富功能。

微软 Azure 云区块链服务

Azure 是微软推出的云计算平台，向用户提供开放的 IaaS 和 PaaS 服务。

Azure 陆续在其应用市场中提供了若干个与区块链相关的服务，分别面向多种不同的区块链底层平台，其中包括以太坊和超级账本 Fabric。

可以在应用市场（https://azuremarketplace.microsoft.com/en-us/marketplace/apps）中搜索 “blockchain” 关键字查看这些服务，如下图所示。

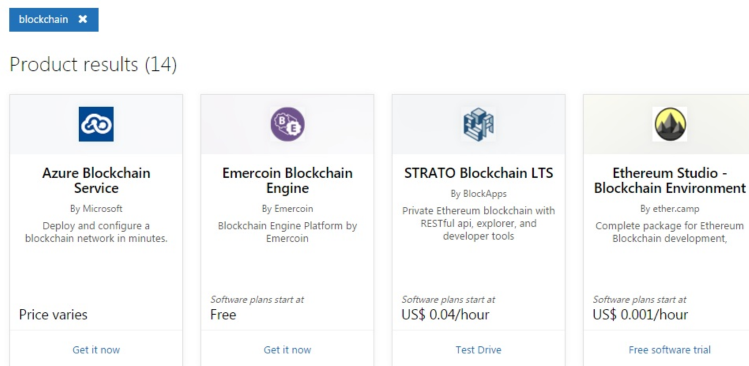


图 *1.15.3.1 - Azure* 上的区块链服务

下面具体介绍其中的 Azure Blockchain Service。

使用服务

使用 Azure 服务，用户可以在几分钟之内在云中部署一个区块链网络。云平台会将一些耗时的配置流程自动化，使用户专注在上层应用方案。

Azure 区块链服务目前支持部署以太坊或超级账本 Fabric 网络。

下面以以太坊为例，在 Azure 的仪表盘中，选择创建 Ethereum Consortium Blockchain 后，输入一些配置选项，则可以开始部署该模拟网络，如下图所示。

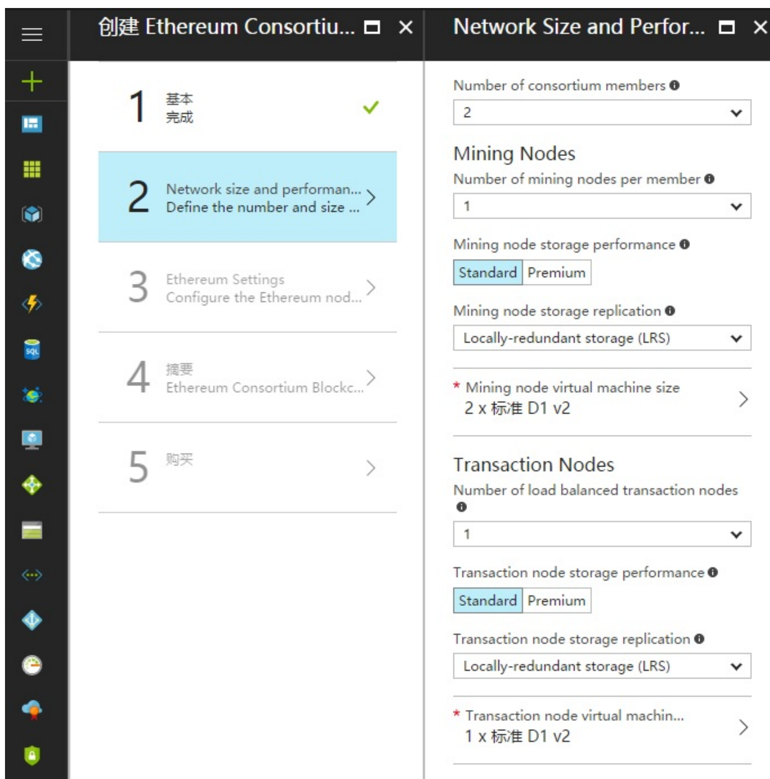


图 *1.15.3.2 - Azure* 区块链配置

部署过程需要几分钟时间。完成后，可进入资源组查看部署结果，如下图所示，成功部署了一个以太坊网络。

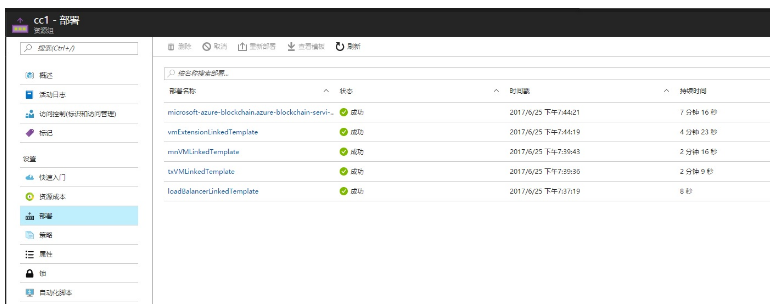


图 *1.15.3.3 - Azure* 区块链部署结果

点击 microsoft-azure-blockchain 开头的链接，可以查看网络的一些关键接口，包括管理网址、RPC 接口地址等。

复制管理网址 ADMIN-SITE 的链接，用浏览器打开，可以进入区块链管理界面。界面中可查看网络各节点信息，也可以新建一个账户，并从 admin 账户向其发送 1000 个以太币。结果如下图所示。

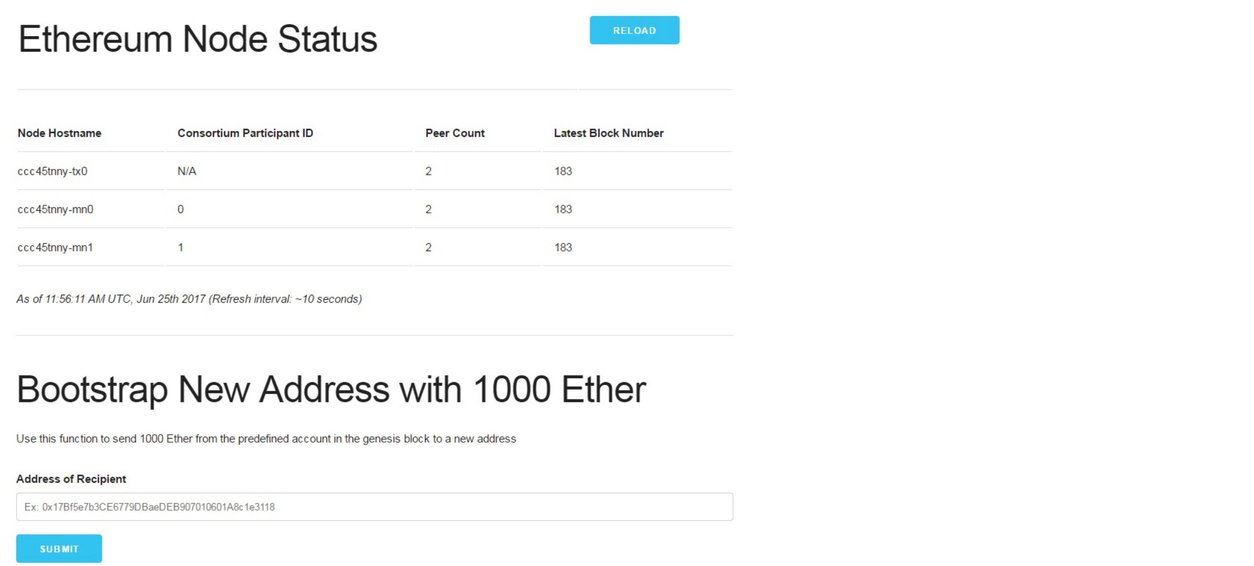


图 *1.15.3.4 - Azure* 区块链管理界面

Azure 云平台提供了相对简单的操作界面，更多的是希望用户通过 RPC 接口地址来访问所部署的区块链示例。用户可以自行通过 RPC 接口与以太坊模拟网络交互，部署和测试智能合约，此处不再赘述。

使用超级账本 Cello 搭建区块链服务

从前面的讲解中可以看到，区块链服务平台能够有效加速对区块链技术的应用，解决企业和开发者进行手动运营管理的负担。但是这些方案都是商业用途，并且只能在线使用。

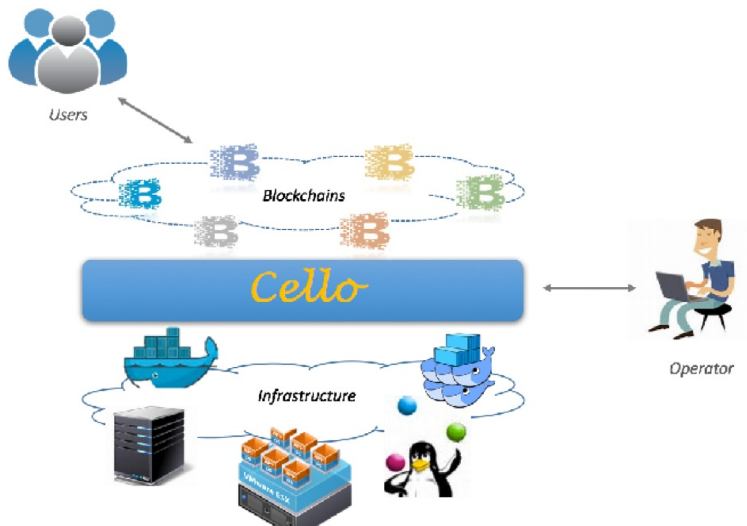


图 *1.15.4.1 - Cello* 典型应用场景

超级账本的 Cello 项目为本地搭建区块链服务管理平台提供了开源的解决方案，可以实现在多种类型的物理资源上实现区块链网络的生命周期管理。

正如 Cello 的名字所蕴意，它就像一把精巧的大提琴，以区块链为琴弦，可以奏出更加动人的乐章。

基本架构和特性

Cello 项目由笔者领导的 IBM 技术团队于 2017 年 1 月贡献到超级账本社区，主要基于 Python 和 Javascript 语言编写。该项目的定位为区块链管理平台，支持部署、运行时管理和数据分析等功能，可以实现一套完整的 BaaS 系统的快速搭建。其基本架构如下图所示。

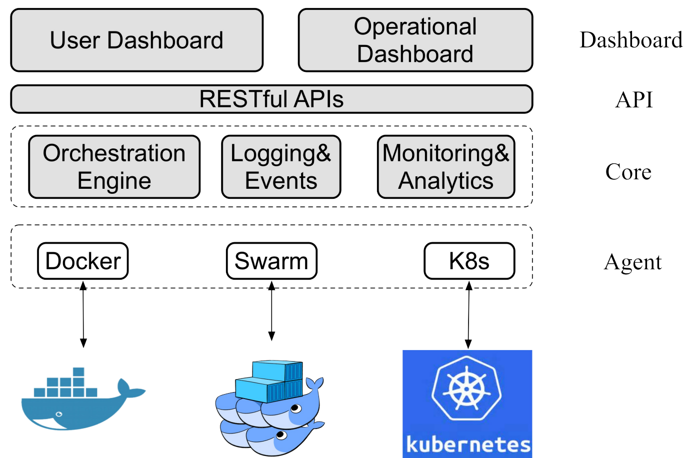


图 *1.15.4.2 - Cello* 基本架构

在实现区块链环境快速部署的同时，Cello 也提供了不少对区块链平台进行运行时管理的特性，这些特性总结如下。

- 管理区块链的全生命周期，包括创建、配置、使用、健康检查、删除等。
- 支持多种基础架构作为底层资源池，包括裸机、虚拟机、容器云（Docker、Swarm、Kubernetes）等。
- 支持多种区块链平台及自定义配置（目前以支持超级账本 Fabric 为主）。
- 支持监控和分析功能，实现对区块链网络和智能合约的运行状况分析。
- 提供可插拔的框架设计，包括区块链平台、资源调度、监控、驱动代理等都很容易引入第三方实现。

下面具体介绍如何以 Docker 主机为资源池，用 Cello 快速搭建一个区块链服务平台。

环境准备

Cello 采用了典型的主从（Master-Worker）架构。用户可以自行准备一个 Master 物理节点和若干个 Worker 节点。

其中，Master 节点负责管理（例如，创建和删除）Worker 节点中的区块链集群，其通过 8080 端口对外提供网页 Dashboard，通过 80 端口对外提供 RESTful API。Worker 节点负责提供区块链集群的物理资源，例如基于 Docker 主机或 Swarm 的方式启动多个集群，作为提供给用户可选的多个区块链网络环境。

下图中展示了一个典型的 Master-Worker 部署拓扑。每个节点默认为 Linux（如 Ubuntu 16.04）服务器或虚拟机。

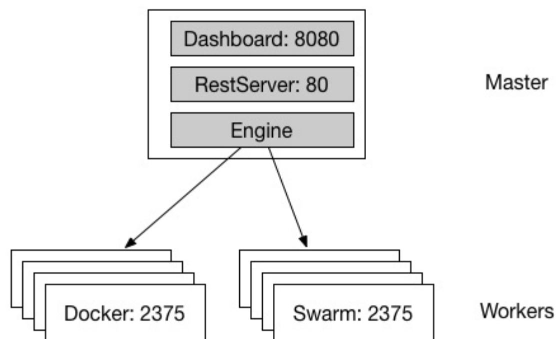


图 *1.15.4.3 - Cello* 部署拓扑示例

为了支持区块链网络，Worker 和 Master 节点需要配备足够的物理资源。例如，如果希望在一个 Worker 节点上能够启动至少 10 个区块链集群，则建议节点配置至少为 8 CPU、16G 内存、100G 硬盘容量。

下载 Cello 源码

Cello 代码的官方仓库在社区的 gerrit 上，并实时同步到 Github 仓库中，读者可以从任一仓库中获取代码。例如通过如下命令从官方仓库下载 Cello 源码。

```
$ git clone http://gerrit.hyperledger.org/r/cello && cd cello
```

配置 Worker 节点

安装和配置 Docker 服务

首先安装 Docker，推荐使用 1.12 或者更新的版本。可通过如下命令快速安装 Docker。

```
$ curl -fsSL https://get.docker.com/ | sh
```

安装成功后，修改 Docker 服务配置。对于 Ubuntu 16.04，更新 `/lib/systemd/system/docker.service` 文件如下。

```
[Service]
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --api-c
ors-header='*' --default-ulimit=nofile=8192:16384 --default-ulimit=nproc=8192:16384"
EnvironmentFile=-/etc/default/docker
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// $DOCKER_OPTS
```

修改后，需要通过如下命令重启 Docker 服务。

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker.service
```

下载 Docker 镜像

对于超级账本 Fabric v1.0 集群所需的镜像，可以使用如下命令进行自动下载。

```
$ cd scripts/worker_node_setup && bash download_images.sh
```

防火墙配置

为了确保 Worker 上的容器可以正常访问，通过如下命令确保主机开启 IP 转发。

```
$ sysctl -w net.ipv4.ip_forward=1
```

同时检查主机的 iptables 设置，确保必要的端口被打开（如 2375、7050~10000 等）。

配置 Master 节点

下载 Docker 镜像

使用如下命令下载运行服务所必要的 Docker 镜像。

其中，python:3.5 镜像是运行 Cello 核心组件的基础镜像；mongo:3.2 提供了数据库服务；yeasy/nginx:latest 提供了 Nginx 转发功能；mongo-express:0.30 镜像是为了调试数据库，可以选择性安装。

```
$ docker pull python:3.5 \
    && docker pull mongo:3.2 \
    && docker pull yeasy/nginx:latest \
    && docker pull mongo-express:0.30
```

安装 Cello 服务

首次运行时，可以通过如下命令对 Master 节点进行快速配置，包括安装 Docker 环境、创建本地数据库目录、安装依赖软件包等。

```
$ make setup
```

如果安装过程没有提示出现问题，则说明当前环境满足了运行条件。如果出现问题，可通过查看日志信息进行定位。

管理 Cello 服务

可以通过运行如下命令来快速启动 Cello 相关的组件服务（包括 dashboard、restserver、watchdog、mongo、nginx 等）。

```
$ make start
```

类似地，运行 `make stop` 或 `make restart` 可以停止或重启全部服务。

若希望重新部署某个特定服务（如 dashboard），可运行如下命令。

```
$ make redeploy service=dashboard
```

运行如下命令可以实时查看所有服务的日志信息。

```
$ make logs
```

若希望查看某个特定服务的日志，可运行如下命令进行过滤，如只查看 watchdog 组件的日志。

```
$ make log service=watchdog
```

使用 Cello 管理区块链

Cello 服务启动后，管理员可以通过 Cello 的 Dashboard 页面管理区块链。

默认情况下，可通过 Master 节点的 8080 端口访问 Dashboard。默认的登录用户名和密码为 `admin:pass` 。

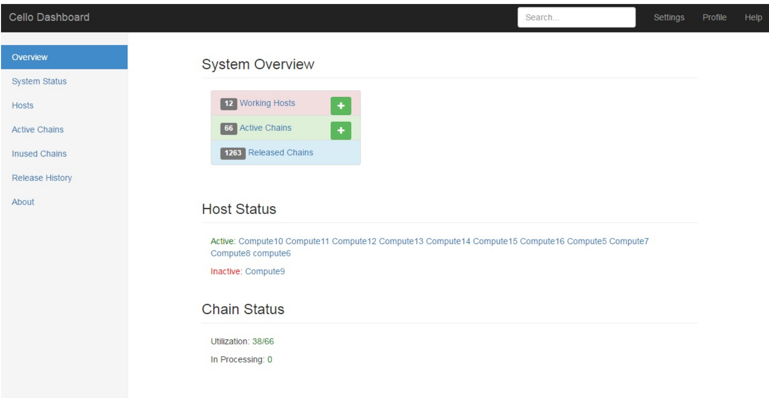


图 *1.15.4.4 - Cello Dashboard*

如图，Dashboard 有多个页面，各页面的功能如下。

页面	功能
Overview	展示系统整体状态
System Status	展示一些统计信息
Hosts	管理所有主机（Worker 节点）
Active Chains	管理资源池中的所有链
Inused Chains	管理正在被用户占用的链
Released History	查看链的释放历史

Hosts 页面

在 Hosts 页面，管理员可以管理所有资源池中已存在的主机，或添加新主机。表格中会显示主机的类型、状态、正在运行的区块链数量、区块链数量上限等。所有设定为 non-schedulable (不会自动分配给用户）的主机会用灰色背景标识，如下图所示。

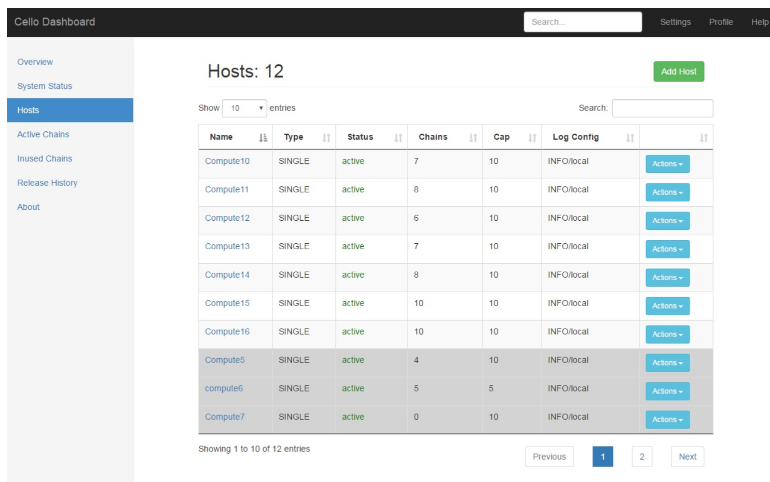


图 *1.15.4.5 - Hosts* 页面

点击一个主机的 Action 下拉菜单，有如下选项可供操作该主机。

- Fillup：将主机运行的区块链数添加至上限。
- Clean：清理主机中所有未被用户占用的链。
- Config：更改主机配置，如名称和链数量上限。
- Reset：重置该主机，只有当该主机没有用户占用的链时可以使用。
- Delete：从资源池中删除该主机。

点击 Hosts 页面的 Add Host 按钮，可以向资源池中添加主机。需要设定该主机的名称、Daemon URL 地址（例如，Worker 节点的 docker daemon 监听地址和端口）、链数量上限、日志配置、是否启动区块链至数量上限、是否可向用户自动分配，如下图所示。

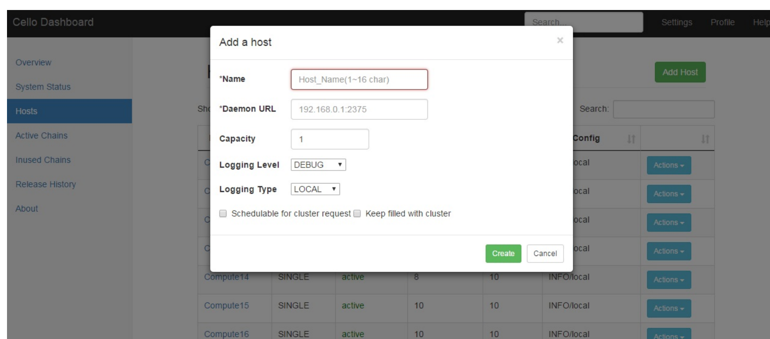


图 *1.15.4.6 -* 添加主机

Active Chains 页面

Active Chains 页面会显示所有正在运行的链，包括链的名称、类型、状态、健康状况、规模、所属主机等信息。正在被用户占用的链会用灰色背景标识，如下图所示。

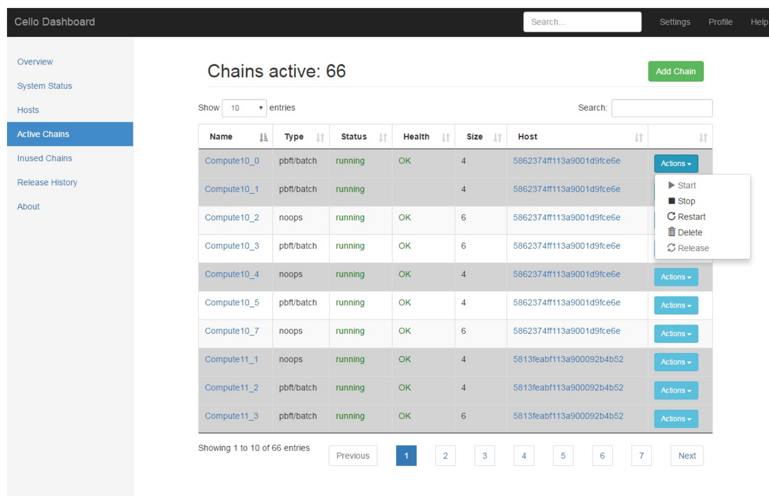


图 *1.15.4.7 - Active Chains* 页面

点击一条链的 Actions 下拉菜单，有如下选项可供操作该链。

- Start：如果这条链处于停止状态，则启动。
- Stop：停止运行中的链。
- Restart：重新启动这条链。
- Delete：删除这条链。
- Release：将占用的链释放，随后会被删除。

点击 Active Chains 页面的 Add Chain 按钮，可以向资源池中添加更多链（如果还有未被占满的主机），如下图所示。

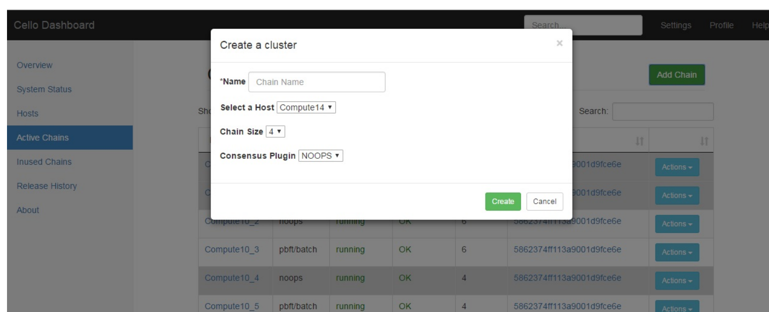


图 *1.15.4.8 -* 添加链

基于 Cello 进行功能扩展

Cello 已经提供了完整的区块链管理功能，并提供了图形界面和 API。

用户可以通过向 Cello 的 Master 节点（默认为 80 端口）发送 RESTful API 来申请、释放区块链，或查看区块链相关信息，如其对外开放的接口，可供用户进行远程交互。RESTful API 的说明可在 Cello 的文档中查阅。

对于区块链服务提供者，可以利用这些 API 为用户呈现友好的区块链申请和操作界面，在 Cello 的基础之上构建和实现更多功能。

本章小结

本章介绍了区块链即服务的概念，阐述了整合云计算技术能够为区块链部署和管理所带来的便捷。接下来提出了区块链服务平台的参考架构，并从功能和性能等实践角度总结了平台设计的考量指标。

本章随后还介绍了业界领先的 IBM Bluemix 和微软 Azure 云上提供的区块链服务。最后讲解了如何使用超级账本 Cello 项目快速搭建一套个性化的区块链服务平台。

区块链技术的普及离不开生态系统和相关工具的成熟，区块链应用的落地同样离不开完善的 DevOps 支持。本章的内容能够给予读者不同的视角，从系统方案的角度出发，思考如何在新技术变革中保持应对变化的敏捷与高效。

性能与评测

过早优化，往往引来各种麻烦。

一项技术究竟能否实用，有两项基本指标十分关键：一是功能的完备；一是性能的达标。

本章将试图对已有区块链技术进行一些评测。所有结果将尽可能保证客观准确，但不保证评测方法是否科学、评测结果是否具备足够参考性。

简介

区块链的平台性能跟很多因素都有关系，特别在实际应用中，根据应用场景的不同和系统设计和使用的不同，可能同一套平台最终在业务体现上会有较大差异。

在这里，仅侧重评测一般意义上的平台性能。

所有给出指标和结果仅供参考，由于评测环境和方案不同，不保证结果的一致性。

生产环境中应用区块链技术请务必进行充分验证评测。

Hyperledger Fabric v0.6 性能评测

环境配置

类型	操作系统	内核版本	CPU(GHz)	内存(GB)
物理机	Ubuntu 14.04.1	3.16.0-71-generic	4x2.0	8

每个集群启动后等待 10s 以上，待状态稳定。

仅测试单客户端、单服务端的连接性能情况。

评测指标

一般评测系统性能指标包括吞吐量（throughput）和延迟（latency）。对于区块链平台系统来说，实际交易延迟包括客户端到系统延迟（往往经过互联网），再加上系统处理反馈延迟（跟不同 consensus 算法关系很大，跟集群之间互联系统关系也很大）。

本次测试仅给出大家最为关注的交易吞吐量（tps）。

结果

query 交易

noops

clients	VP Nodes	iteration	tps
1	1	2000	195.50
1	4	2000	187.09

pbft:classic

clients	VP Nodes	iteration	tps
1	4	2000	193.05

pbft:batch

clients	VP Nodes	batch size	iteration	tps
1	4	2	2000	193.99
1	4	4	2000	192.49
1	4	8	2000	192.68

pbft:sieve

clients	VP Nodes	iteration	tps
1	4	2000	192.86

invoke 交易

noops

clients	VP Nodes	iteration	tps
1	1	2000	298.51
1	4	2000	205.76

pbft:classic

clients	VP Nodes	iteration	tps
1	4	2000	141.34

pbft:batch

clients	VP Nodes	batch size	iteration	tps
1	4	2	2000	214.36
1	4	4	2000	227.53
1	4	8	2000	237.81

pbft:sieve

clients	VP Nodes	iteration	tps
1	4	2000	253.49*

注：*sieve* 算法目前在所有交易完成后较长时间内并没有取得最终的结果，出现大量类似*“vp0_1 | 07:49:26.388 [consensus/obcpbft] main -> WARN 23348 Sieve replica 0 custody expired, complaining:*

*3kwyMkdCSL4rbajn65v+iYWyJ5aqagXvRR9QU8qezpAZXY4y6uy2MB31SGaAiaSyPMM77
TYADdBmAaZveM38zA==”*警告信息。

结论

单客户端连接情况下，tps 基本在 190 ~ 300 范围内。

小结

附录

术语

通用术语

- Blockchain（区块链）：基于密码学的可实现信任化的信息存储和处理的结构和技术。
- Byzantine Failure（拜占庭错误）：指系统中存在除了消息延迟或不可送达的故障以外的错误，包括消息被篡改、节点不按照协议进行处理等，潜在地会对系统造成针对性的破坏。
- CDN：Content Delivery Network，内容分发网络。利用在多个地理位置预先配置的缓存服务器，自动从距离近的缓存服务器进行对请求的响应，以实现资源的快速分发。
- Consensus（共识）：分布式系统中多个参与方对某个信息达成一致，多数情况下为对发生事件的顺序达成一致。
- Decentralization（去中心化）：无需一个独立第三方的中心机构存在，有时候也叫多中心化。
- Distributed（分布式）：非单体中央节点的实现，通常由多个个体通过某种组织形式联合在一起，对外呈现统一的服务形式。
- Distributed Ledger（分布式账本）：由多家联合维护的去中心化（或多中心化）的账本记录平台。
- DLT：Distributed Ledger Technology，分布式账本技术。包括区块链、权限管理等在内的实现分布式账本的技术。
- DTCC：Depository Trust and Clearing Corporation，存托和结算公司。全球最大的金融交易后台服务机构。
- Fintech：Financial Technology，跟金融相关的（信息）技术。
- Gossip：一种 P2P 网络中多个节点之间进行数据同步的协议，如随机选择邻居进行转发。
- LDAP：Lightweight Directory Access Protocol，轻量级目录访问协议，是一种为查询、搜索业务而设计的分布式数据库协议，一般具有优秀的读性能，但写性能往往较差。
- Market Depth（市场深度）：衡量市场承受大额交易后汇率的稳定能力，例如证券交易市场出现大额交易后价格不出现大幅波动。
- MTBF：Mean Time Between Failures，平均故障间隔时间，即系统可以无故障运行的预期时间。
- MTTR：Mean Time to Repair，平均修复时间，即发生故障后，系统可以恢复到正常运行的预期时间。
- MVCC：Multi-Version Concurrency Control，多版本并发控制。数据库领域的技术，通过引入版本来实现并发更新请求的乐观处理，当更新处理时数据版本跟请求中注明版本不一致时则拒绝更新。发生更新成功则将数据的版本加一。
- Non-validating Peer（非验证节点）：不参与账本维护，仅作为交易代理响应客户端的请求，并对交易进行一些基本的有效性检查，之后转发给验证节点。

- P2P：点到点的通信网络，网络中所有节点地位均等，不存在中心化的控制机制。
- SLA/SLI/SLO：Service Level Agreement/Indicator/Objective，分别描述服务可用性对用户的承诺，功能指标和目标值。
- SWIFT：Society for Worldwide Interbank Financial Telecommunication，环球银行金融电信协会，运营世界金融电文网络，服务银行和金融机构。
- Turing-complete（图灵完备）：指一个机器或装置能用来模拟图灵机（现代通用计算机的雏形）的功能，图灵完备的机器在可计算性上等价。
- Validating Peer（验证节点）：维护账本的核心节点，参与一致性维护、对交易的验证和执行。更进一步可以划分为 Endorser、Committer 等多种角色。

密码学与安全相关

- ASN.1：Abstract Syntax Notation One，定义了描述数据的表示、编码、传输、解码的一套标准，被广泛应用在计算机、通信和安全领域。
- CA：Certificate Authority，负责证书的创建、颁发，在 PKI 体系中最为核心的角色。
- CRL：Certification Revocation List，证书吊销列表，包含所撤销的证书列表。
- CSR：Certificate Signing Request，证书签名申请，包括通用名、名称、主机、生成私钥算法和大小、CA 配置和序列号等信息，用来发给 CA 服务以颁发签名的证书。
- DER：Distinguished Encoding Rules，ASN.1 中定义的一种二进制编码格式，可以用来保存证书或密钥内容。
- Genesis Block：创世区块，区块链的第一个区块，一般用于初始化，不带有交易信息。
- Hash：哈希算法，任意长度的二进制值映射为较短的固定长度的二进制值的算法。
- IES：Integrated Encryption Scheme，集成加密机制，一种混合加密机制，可以应对选择明文攻击（可以获知任意明文和对应密文）情况下的攻击。包括 DLIES（基于离散对数）和 ECIES（基于椭圆曲线）两种实现。
- Nonce：密码学术语，表示一个临时的值，多为随机字符串。
- OCSP：Online Certificate Status Protocol，在线证书状态协议，通过查询服务来在线确认证书的状态（如是否撤销）。RFC 2560 中定义。
- PKCS：Public-Key Cryptography Standards，公钥密码标准，由 RSA 实验室提出，定义了利用 RSA 算法和相关密码学技术来实现安全的系列规范，目前包括 15 个不同领域的规范。最早的版本在 1991 年提出，目前最新版本为 2012 年提出的 2.2 版本。
- PEM：Privacy Enhanced Mail，用来保存证书和密钥的一种编码格式，RFC 1421-1424 中定义。
- PKI：Public Key Infrastructure，基于公钥体系的安全基础架构。
- SM：国家商用密码算法，2010 年以来陆续由国家密码管理局发布的相关标准和规范，主要包括：SM2（基于椭圆曲线密码的公钥密码算法标准）、SM3（Hash 算法标准）、SM4（基于分组加密的对称密码算法标准）、SM9（基于身份的数字证书体系）。

比特币、以太坊相关术语

* Bitcoin（比特币）：最早由中本聪提出和实现的基于区块链思想的数字货币技术。
* DAO：Decentralized Autonomous Organization，分布式自治组织，基于区块链的按照智能合约联系起来的松散自治群体。
* Lightning Network（闪电网络）：通过链外的微支付通道来增大交易吞吐量的技术。
* Mining（挖矿）：通过暴力尝试来找到一个字符串，使得它加上一组交易信息后的 Hash 值符合特定规则（例如前缀包括若干个 0），找到的人可以宣称新区块被发现，并获得系统奖励的数字货币。
* Miner（矿工）：参与挖矿的人或组织。
* Mining Machine（矿机）：专门为数字货币挖矿而设计的设备，包括基于软件、GPU、FPGA、专用芯片等多种实现。
* Mining Pool（矿池）：采用团队协作方式来集中算力进行挖矿，对产出的数字货币进行分配。
* PoS：Proof of Stake，股份持有证明，拥有代币或股权越多的用户，挖到矿的概率越大。
* PoW：Proof of Work，工作量证明，在一定难题前提下求解一个 SHA256 的 Hash 问题。
* Smart Contract（智能合约）：运行在区块链上的提前约定的合同；
* Sybil Attack（女巫攻击）：少数节点通过伪造或盗用身份伪装成大量节点，进而对分布式系统系统进行破坏。

超级账本相关术语

* Anchor（锚定）：一般指作为刚启动时候的初始联络元素或与其它结构的沟通元素。如刚加入一个通道的节点，需要通过某个锚点节点来快速获取通道内的情况（如其它节点的存在信息）。
* Auditability（审计性）：在一定权限和许可下，可以对链上的交易进行审计和检查。
* Block（区块）：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。
* Blockchain（区块链）：由多个区块链接而成的链表结构，除了初始区块，每个区块头部都包括前继区块内容的 Hash 值。
* Chaincode（链码）：区块链上的应用代码，扩展自“智能合约”概念，支持 Golang、Nodejs 等语言，多为图灵完备。
* Channel（通道）：Fabric 网络上的私有隔离机制。通道中的链码和交易只有加入该通道的节点可见。同一个节点可以加入多个通道，并为每个通道内容维护一个账本。
* Committer（提交节点）：一种 Peer 节点角色，负责对 Orderer 排序后的交易进行检查，选择合法的交易执行并写入存储。
* Commitment（提交）：提交节点完成对排序后交易的验证，将交易内容写到区块，并更新世界状态的过程。
* Confidentiality（保密）：只有交易相关方可以看到交易内容，其它人未经授权则无法看到。
* Endorser（推荐节点或背书节点）：一种 Peer 节点角色，负责检验某个交易是否合法，

是否愿意为之背书、签名。

- Endorsement：背书过程。按照链码部署时候的背书策略，相关 Peer 对交易提案进行模拟和检查，决策是否为之背书。如果交易提案获得了足够多的背书，则可以构造正式交易进行进一步的共识。
- Invoke（调用）：一种交易类型，对链码中的某个方法进行调用，一般需要包括调用方法和调用参数。
- Ledger（账本）：包括区块链结构（带有所有的交易信息）和当前的世界状态（world state）。
- Member（成员）：代表某个具体的实体身份，在网络中拥有自己的根证书。节点和应用都必须属于某个成员身份。同一个成员可以在同一个通道中拥有多个 Peer 节点，其中一个为 Leader 节点，代表成员与排序节点进行交互，并分发排序后的区块给属于同一成员的其它节点。
- MSP（Member Service Provider，成员服务提供者）：抽象的实现成员服务（身份验证，证书管理等）的组件，实现对不同类型的成员服务的可拔插支持。
- Orderer（排序节点）：共识服务角色，负责排序看到的交易，提供全局确认的顺序。
- Permissioned Ledger（带权限的账本）：网络中所有节点必须是经过许可的，非许可过的节点则无法加入网络。
- Privacy（隐私保护）：交易员可以隐藏交易的身份，其它成员在无特殊权限的情况下，只能对交易进行验证，而无法获知身份信息。
- System Chain（系统链）：由对网络中配置进行变更的配置区块组成，一般可以用来作为组成网络成员们形成的联盟约定。
- Transaction（交易）：执行账本上的某个函数调用或者部署、更新链码。调用的具体函数在链码中实现。
- Transactor（交易者）：发起交易调用的客户端。
- World State（世界状态）：即最新的全局账本状态。Fabric 用它来存储历史交易发生后产生的最新的状态，可以用键值或文档数据库实现。

常见问题

通用问题

问：区块链是谁发明的，有什么特点？

答：区块链相关的思想最早是比特币的发明者-中本聪（化名）在论文中提出（但没有明确定义）作为比特币网络的核心支持技术。自那以后，区块链技术逐渐脱离比特币网络，成为一种通用的可以支持分布式记账能力的底层技术，具有去中心化和加密安全等特点。

问：区块链和比特币是什么关系？

答：比特币是基于区块链技术的一种数字现金（cash）应用；区块链技术最早在比特币分布式系统中得到应用和验证，确保了比特币系统在 2009 年上线后，在完全自治情况下可以正常运转。

问：区块链和分布式数据库是什么关系？

答：两者定位完全不同。分布式数据库是解决高可用和可扩展场景下的数据存储问题；区块链则是在多方（无须中心化中介角色存在）之间提供一套可信的记账和合约履行机制。

问：区块链有哪些种类？

答：根据部署场景公开程度，可以分为公有链（Public Chain）、联盟链（Consortium Chain）和私有链（Private Chain）；从功能上看，可以分为以支持数字货币为主的数字货币区块链（如比特币网络）、支持智能合约的通用区块链（如以太坊网络）、面向复杂商业应用场景支持权限管理的分布式账本平台（如超级账本）。

问：（公有链情况下）区块链是如何保证没有人作恶的？

答：区块链并没有试图保障每一个人都不作恶，每个参与者都默认在最长的链上进行扩展。当某个作恶者尝试延续一个非法链的时候，实际上在跟所有的“非作恶”者进行竞争。因此，当作恶者超过一半（还要保持选择一致）时，在概率意义上才能破坏规则。而代价是一旦延续失败，所有付出的资源（例如算力）都将浪费掉。

问：区块链的智能合约应该怎么设计？

答：首先，智能合约类似其它应用程序，在架构上即可以采取 monolithic 的方式（一个合约针对一个具体商业应用，功能完善而复杂），也可以采取 microservice 的方式（即每个合约功能单一，多个合约一起构建应用）。 选择哪种模式根本上取决于其上商业应用的特点。从灵活性角度，推荐适当对应用代码进行切分，划分到若干个智能合约，尽量保持智能合约的可复用性。

问：如何查看 **PEM** 格式证书内容？

答：可以通过如下命令转换证书内容进行输出：`openssl x509 -noout -text -in <ca_file>`；例外，还可以通过如下命令来快速从证书文件中提取所证明的公钥内容：`openssl x509 -noout -pubkey -in <ca_file>`。

问：已知私钥，如何生成公钥？

答：对于椭圆曲线加密算法，可以通过如下命令生成公钥：`openssl ec -pubout -outform PEM -in <private_key>`。

问：如何校验某证书是否被根证书签名？

答：已知根证书文件 和待验证证书文件 情况下，可以使用如下命令进行验证：`openssl verify -CAfile <root_cafile> <ca_to_verify>`。

问：为何 **Hash** 函数将任意长的文本映射到定长的摘要，很少会发生冲突？

答：像 SHA-1 这样的 Hash 函数可以将任意长的文本映射到相对很短的定长摘要。从理论上讲，从一个很大的集合映射到一个小的集合上必然会出现冲突。Hash 函数之所以很少出现冲突的原因在于虽然输入的数据长度可以很大，但其实人类产生的数据并非全空间的，这些数据往往是相对有序（低熵值）的，实际上也是一个相对较小的集合。当然，这个集合自身可能比输出的结果要大，但这个冲突的概率远没有输入是全空间集合时那么夸张。

比特币、以太坊相关

问：比特币区块链为何要设计为每 **10** 分钟才出来一个块，快一些不可以吗？

答：这个主要是从公平的角度，当某一个新块被计算出来后，需要在全球的比特币网络内公布。临近的矿工将最先拿到消息并开始新一轮的计算，较远的矿工则较晚得到通知。最坏情况下，可能需要数十秒的延迟。为尽量确保矿工们都处在同一起跑线上，这个时间不能太短。但太长了又会导致每个交易的“最终”确认时间过长，目前看，10 分钟左右是一个相对合适的折中。另外，也是从存储代价的角度，让拥有不太大存储的普通节点可以参与到网络的维护。

问：比特币区块链每个区块大小为何是 **1 MB**，大一些不可以吗？

答：这个也是折中的结果。区块产生的平均时间间隔是固定的 10 分钟，大一些，意味着发生交易的吞吐量可以增加，但节点进行验证的成本会提高（Hash 处理约为 100 MB/s），同时存储整个区块链的成本会快速上升。区块大小为 1 MB，意味着每秒可以记录 1 MB/(10*60)=1.7 KB 的交易数据，而一般的交易数据大小在 0.2 ~ 1 KB。

实际上，之前比特币社区也曾多次讨论过改变区块大小的提案，但都未被最终接受。

问：以太坊网络跟比特币网络有何关系？

答：以太坊网络所采用的区块链结构，源于比特币网络。基于同样设计原理上，以太坊提出了许多改善设计，包括支持更灵活的智能合约、支持除了 PoW 之外的更多共识机制（尚未实现）等。

超级账本项目

问：超级账本项目与传统公有区块链有何不同？

答：超级账本是首个面向联盟链场景的开源项目，在这种场景下，参与账本的多方存在一定的信任前提，并十分看重对接入账本各方的权限管理、审计功能、传输数据的安全可靠等特性。超级账本在考虑了商业网络的这些复杂需求后，提出了创新的架构和设计，是首个在企业应用场景中得到大规模部署和验证的开源项目。

问：区块链最早是公有链形式，为何现在联盟链在很多场景下得到更多推崇？

答：区块链技术出现以前，中心化的信任机制可以实现很高的性能和便捷的监管，但一旦中心机制出现故障，则导致系统的信任前提发生破坏。区块链技术可以提供无中介化情况下的信任保障。公有链情况下，任何人都可以参与监督，可以实现信任的最大化，但随之而来带来包括性能低下、缺乏监管等问题。

联盟链在两者之间取得了平衡。非中心化的联盟共识，让系统可信任度以指数形式增加；同时，联盟形成的信任前提，可以在不影响信任的情况下实现更优化的性能，并支持权限管理。这对复杂应用场景特别企业场景可以提供更好的支持。

问：采用 **BFT** 类共识算法时，节点掉线后重新加入网络，出现无法同步情况？

答：这是某些算法设计导致的情况。掉线后的节点重新加入到网络中，其视图（View）会领先于其它节点。其它节点正常情况下不会发生视图的变更，发生的交易和区块内容不会同步到掉线节点。出现这种情况，可以有两种解决方案：一个是强迫其它节点出现视图变更，例如也发生掉线或者在一段时间内强制变更；另一种情况是等待再次产生足够多的区块后触发状态追赶。

问：超级账本 **Fabric** 里的安全性和隐私性是如何保证的？

答：首先，Fabric 1.0 及往后的版本提供了对多通道的支持，不同通道之间的链码和交易是不可见的，即交易只会发送到该通道内的 Peer 节点。此外，在进行背书阶段，客户端可以根据背书策略来选择性的发送交易到通道内的某些特定 Peer 节点。更进一步的，用户可以对交易的内容进行加密（基于证书的权限管理）或 Hash 处理，同时，只有得到授权的节点或用户才能访问到交易。另外，排序节点无须访问到交易内容，因此，可以选择不将完整交易（对交易输入数据进行隐藏，或者干脆进行加密或 Hash 处理）发送到排序节点。最后，所有数据在传输过程中可以通过 TLS 来进行安全保护。许多层级的保护需要配合使用来获得不同层级的安全性。

实践过程中，也需要对节点自身进行安全保护，通过防火墙、IDS 等防护对节点自身的攻击；另外可以通过审计和分析系统对可疑行为进行探测和响应。

相关企业和组织

排名不分先后，大部分信息来自互联网，不保证信息准确性，如有修改意见，欢迎联系。

国际

企业

- IBM: 贡献区块链平台代码到 HyperLedger 项目，推动区块链产业发展，跟多家银行和企业进行区块链项目合作。
- DTCC:贡献区块链代码到 HyperLedger 项目。
- Circle：基于区块链的支付应用公司，已获得 6000 万美元 D 轮投资，投资者包括 IDG、百度、中金甲子、广大投资等，目前年交易额超过 10 亿美金。
- Consensus：区块链创业团队，试图打造区块链平台技术和应用支撑，获得多家投资。

组织

- R3 CEV：创立于 2015 年 9 月，总部位于纽约的金融联盟组织，专注于研究和评估基于区块链的金融技术解决方案，由 40 多家国际金融机构组成，包括 Citi、BOA、高盛、摩根、瑞银、IBM、微软等。R3 开源技术已经 宣布 加入 HyperLedger 项目。
- [HyperLedger 社区]（https://hyperledger.org）：创立于 2015 年 12 月的技术社区，由 Linux 基金会管理，包括 IBM、Accenture、Intel、J.P.Morgan、R3、DAH、DTCC、FUJITSU、HITACHI、SWIFT、Cisco 等多家企业参与成立，试图打造面向企业应用场景的分布式账本平台。
- Ethereum 社区: 围绕以太坊区块链平台的开放社区。
- DAO： Distributed Autonomous Organization，基于以太坊平台的公募基金(众筹)组织，或去中心化的风投。众筹资金超过 1.6 亿美金。

国内

学术界

- 清华大学
- 中科院
- 上海交通大学

企业

- 中国电信：研究区块链相关技术，包括去中心化共享经济平台等。
- 世纪互联：投资区块链技术团队，牵头成立“中关村区块链产业联盟”。
- 银联：关注区块链相关技术，尝试引入基于区块链的银行业积分系统。
- 能链：专注于能源产品相关的区块链应用。
- 恒生电子：2016 年牵头成立“金链盟”，希望通过区块链技术为金融行业提供更简单的产品。
- 布比：主要关注数字资产管理的技术型创业企业，区块链相关平台和产品。
- 小蚁：主要关注对资产和权益进行数字化，2014年于上海组建成立。
- 火币：国内较大的比特币交易代理平台。
- BeLink：关注保险行业积分系统，主要产品为数贝荷包。
- BitSe：主要产品为唯链（Vechain），面向物品防伪追踪、数字版权管理相关。
- 万向集团：投资多家区块链创业团队，致力于推动产业发展。

组织

- 中关村区块链产业联盟：2016 年 2 月 3 日成立于北京，由世纪互联联合清华大学、北京邮电大学等高校、中国通信学会、中国联通研究院等运营商，及集佳、布比网络等公司发起；
- ChinaLedger：2016 年 4 月成立于上海，成员包括中证机构间报价系统股份有限公司、中钞信用卡产业发展有限公司北京智能卡技术研究院、万向区块链实验室、浙江股权交易中心、深圳招银前海金融资产交易中心、厦门国际金融资产交易中心、大连飞创信息技术有限公司、通联支付网络服务股份有限公司、上海矩真金融信息服务有限公司、深圳瀚德创客金融投资有限公司、乐视金融等；
- 金融区块链合作联盟（金链盟）：2016 年 5 月 31 日成立于深圳，包括平安银行、恒生电子、京东金融、腾讯微众银行、华为、南方基金、国信证券、安信证券、招商证券、博时基金等 25 家公司与机构。

ProtoBuf 与 gRPC

ProtoBuf 是一套接口描述语言（Interface Definition Language，IDL），类似 Apache 的 Thrift。

相关处理工具主要是 protoc，基于 C++ 语言实现。

用户写好 `.proto` 描述文件，之后便可以使用 protoc 自动编译生成众多计算机语言（C++、Java、Python、C#、Golang 等）的接口代码。这些代码可以支持 gRPC，也可以不支持。

gRPC 是 Google 开源的 RPC 框架和库，已支持主流计算机语言。底层通信采用 HTTP2 协议，比较适合互联网场景。gRPC 在设计上考虑了跟 ProtoBuf 的配合使用。

两者分别解决不同问题，可以配合使用，也可以分开单独使用。

典型的配合使用场景是，写好 `.proto` 描述文件定义 RPC 的接口，然后用 protoc（带 gRPC 插件）基于 `.proto` 模板自动生成客户端和服务端的接口代码。

ProtoBuf

需要工具主要包括：

- 编译工具：protoc，以及一些官方没有带的语言插件；
- 运行环境：各种语言的 protobuf 库，不同语言有不同的安装来源；

语法类似 C++ 语言，可以参考 ProtoBuf 语言规范：https://developers.google.com/protocol-buffers/docs/proto。

比较核心的，`message` 是代表数据结构（里面可以包括不同类型的成员变量，包括字符串、数字、数组、字典……），`service` 代表 RPC 接口。变量后面的数字是代表进行二进制编码时候的提示信息，1~15 表示热变量，会用较少的字节来编码。另外，支持导入。

默认所有变量都是可选的（optional），repeated 则表示数组。主要 service rpc 接口接受单个 message 参数，返回单个 message。如下所示。

```
syntax = "proto3";
package hello;

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
  repeated int32 number=4;
}

service HelloService {
  rpc SayHello(HelloRequest) returns (HelloResponse){}
}
```

编译最关键的参数是输出语言格式参数，例如，python 为 `--python_out=OUT_DIR` 。

一些还没有官方支持的语言，可以通过安装 protoc 对应的 plugin 来支持。例如，对于 Go 语言，可以安装

```
$ go get -u github.com/golang/protobuf/{protoc-gen-go,proto} // 前者是 plugin；后者是 go
 的依赖库
```

之后，正常使用 `protoc --go_out=./ hello.proto` 来生成 hello.pb.go，会自动调用 `protoc-gen-go` 插件。

ProtoBuf 提供了 `Marshal/Unmarshal` 方法来将数据结构进行序列化操作。所生成的二进制文件在存储效率上比 XML 高 3~10 倍，并且处理性能高 1~2 个数量级。

gRPC

相关工具主要包括：

- 运行时库：各种不同语言有不同的安装方法，主流语言的包管理器都已支持。
- protoc，以及 gRPC 插件和其它插件：采用 ProtoBuf 作为 IDL 时，对 .proto 文件进行编译处理。

类似其它 RPC 框架，gRPC 的库在服务端提供一个 gRPC Server，客户端的库是 gRPC Stub。典型的场景是客户端发送请求，同步或异步调用服务端的接口。客户端和服务端之间的通信协议是基于 HTTP2 的 gRPC 协议，支持双工的流式保序消息，性能比较好，同时也很轻。

采用 ProtoBuf 作为 IDL，则需要定义 service 类型。生成客户端和服务端代码。用户自行实现服务端代码中的调用接口，并且利用客户端代码来发起请求到服务端。一个完整的例子可以参考 https://github.com/grpc/grpc-go/blob/master/examples/helloworld 。

以上面 proto 文件为例，需要执行时添加 gRPC 的 plugin：

```
$ protoc --go_out=plugins=grpc:. hello.proto
```

gRPC 更多原理可以参考官方文档：http://www.grpc.io/docs。

生成服务端代码

服务端相关代码如下，主要定义了 HelloServiceServer 接口，用户可以自行编写实现代码。

```
type HelloServiceServer interface {
        SayHello(context.Context, *HelloRequest) (*HelloResponse, error)
}

func RegisterHelloServiceServer(s *grpc.Server, srv HelloServiceServer) {
        s.RegisterService(&_HelloService_serviceDesc, srv)
}
```

用户需要自行实现服务端接口，代码如下。

比较重要的，创建并启动一个 gRPC 服务的过程：

- 创建监听套接字：`lis, err := net.Listen("tcp", port)`；
- 创建服务端：`grpc.NewServer()`；
- 注册服务：`pb.RegisterHelloServiceServer()`；
- 启动服务端：`s.Serve(lis)`。

```
type server struct{}

// 这里实现服务端接口中的方法。
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, e
rror) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

// 创建并启动一个 gRPC 服务的过程：创建监听套接字、创建服务端、注册服务、启动服务端。
func main() {
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterHelloServiceServer(s, &server{})
    s.Serve(lis)
}
```

编译并启动服务端。

生成客户端代码

生成的 go 文件中客户端相关代码如下，主要和实现了 HelloServiceClient 接口。用户可以通过 gRPC 来直接调用这个接口。

```
type HelloServiceClient interface {
        SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error)
}

type helloServiceClient struct {
        cc *grpc.ClientConn
}

func NewHelloServiceClient(cc *grpc.ClientConn) HelloServiceClient {
        return &helloServiceClient{cc}
}

func (c *helloServiceClient) SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error) {
        out := new(HelloResponse)
        err := grpc.Invoke(ctx, "/hello.HelloService/SayHello", in, out, c.cc, opts...)

        if err != nil {
                return nil, err
        }
        return out, nil
}
```

用户直接调用接口方法：创建连接、创建客户端、调用接口。

```
func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    c := pb.NewHelloServiceClient(conn)

    // Contact the server and print out its response.
    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("Greeting: %s", r.Message)
}
```

编译并启动客户端，查看到服务端返回的消息。

参考资源链接

论文

- L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978.
- M Pease, R Shostak, L Lamport. Reaching Agreement in the Presence of Faults. Journal of the ACM, 1980, 27(2): 228-234.
- M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” J. ACM, vol. 32, no. 2, pp. 374–382, 1985.
- L. Lamport, “The Part-Time Parliament,” ACM Trans. Comput. Sys-tems, vol. 16, no. 2, pp. 133–169, 1998.
- M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance,” Proc. Symp. Oper. Syst. Des. Implement., no. February, pp. 1–14, 1999.
- 中本聪，"Bitcoin: A Peer-to-Peer Electronic Cash System（比特币：一种点对点的电子现金系统）"，https://bitcoin.org/bitcoin.pdf，2008. A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, “Enabling Blockchain Innovations with Pegged Sidechains,” pp. 1–25, 2014.
- T. D. Joseph Poon, “The Bitcoin Lightning Network: Scalable Off-Chain Payments, http://lightning.network/lightning-network-paper.pdf,” pp. 1–59, 2016.
- Gentry C., Halevi S.，"Implementing Gentry’s Fully-Homomorphic Encryption Scheme". In: Paterson K.G. (eds) Advances in Cryptology – EUROCRYPT 2011. EUROCRYPT 2011. Lecture Notes in Computer Science, vol 6632. Springer, Berlin, Heidelberg.
- van Dijk M., Gentry C., Halevi S., Vaikuntanathan V., "Fully Homomorphic Encryption over the Integers". In: Gilbert H. (eds) Advances in Cryptology – EUROCRYPT 2010. EUROCRYPT 2010. Lecture Notes in Computer Science, vol 6110. Springer, Berlin, Heidelberg.
- López-Alt, Adriana, Eran Tromer, and Vinod Vaikuntanathan. "On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption.". Proceeding STOC '12 Proceedings of the forty-fourth annual ACM symposium on Theory of computing, Pages 1219-1234.
- I. Miers, C. Garman, M. Green, and A. D. Rubin, “Zerocoin: Anonymous distributed e-cash from bitcoin,” Proc. - IEEE Symp. Secur. Priv., pp. 397–411, 2013.
- F. Reid and M. Harrigan, “An analysis of anonymity in the bitcoin system,” Secur. Priv. Soc. Networks, pp. 197–223, 2013.
- K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, “Formal Verification of Smart Contracts,” 2016.

项目网站

- 比特币项目官方网站：https://bitcoin.org/。
- blockchain.info：比特币信息统计网站。
- bitcoin.it：比特币 wiki，相关知识介绍。
- 以太坊项目官方网站：https://www.ethereum.org。
- 以太坊网络的状态统计：https://etherchain.org/
- 超级账本项目官方网站：https://hyperledger.org;
- 超级账本 Docker 镜像：https://hub.docker.com/r/hyperledger/。

培训课程

- Bitcoin and Cryptocurrency Technologies, Princeton University。

区块链即服务

- IBM Bluemix BaaS： https://console.ng.bluemix.net/catalog/services/blockchain/。
- 微软 Azure BaaS：https://azure.microsoft.com/en-us/solutions/blockchain。